HARVEY MUDD
COLLEGE

ACM UNIX Handbook

March 17, 2006

Written By
Mark Kegel

Contents

Before We Begin...

■ Preface

Welcome to CS60! While taking CS60 you'll be introduced to many different aspects of computer science, all of which I hope you find intriguing and exciting. If you decide to continue and take further courses in computer science, which I hope you do, you'll become a very proficient programmer. One thing that you probably wouldn't have encountered before this manual, however, was a succinct introduction to UNIX. In fact you probably haven't even encountered UNIX before! This manual is here to help guide you as you learn to use UNIX, and more importantly, how to program well. However, regardless of major, I hope that find this manual useful, worthwhile, and demystifying when UNIX problems come your way.

■ Acknowledgments

This book could not have been made possible without the help of some very dedicated people. I would like to thank (I'll fill this in later).

■ Why Read This Manual

One crucial aspect of your CS education at Mudd is your introduction to the UNIX command line, and creating programs that operate correctly from the command line, or at least well, within that context. While most students are likely to be familiar and proficient with the graphical interfaces of Windows and Mac OS X, command line familiarity is not something most students have, or are expected to have. As such, this manual is meant to help ease you, the student, into a command line environment, and give you the tools and knowledge to be an efficient and powerful UNIX user.

The other purpose is of this manual is to introduce some of the most important aspects of programming. This will not only make you into a good coder, but a good UNIX programmer as well. This manual should also be very useful to those students taking later courses in the CS sequence, like CS70 and CS105.

■ Who This Manual Is For

Quite simply put this book is aimed at someone who has little, to no, experience with UNIX, and from personal experience, this is most students. I do hope, though, that more experienced students will find this book useful, either that, or a good paper weight. I've included some of my own experiences with UNIX in the text below, which I hope helps explain *why* I wrote this manual.

I encountered UNIX for the first time while taking CS60, much like many of you, and at first I found it incredibly intimidating and very frightening. The commands were esoteric and kind of random, *emacs* was just bizarre, and there was no consistent form of help that I could find (manpages? huh?). We were given a sheet of some of the more useful commands, *ls* and its ilk, and then let go with no guidance. I went through the whole of CS60 not knowing how to background a process. *ssh* was not something I regularly used until CS70. And X-Forwarding? That came much, much later.

During the first semester of my sophomore year at Mudd I installed Linux with the help of a friend, but it was not until Winter Break that I could even install Linux myself without help. I spent that break learning as much as I could about Linux and how to use the command line. I would spend the next year learning as much about Linux and UNIX as I could.

During the spring of 2005 I approached Prof. Mike Erlinger about having the CS Department teach a course on using UNIX. After much nagging, begging, and pleading I came to the conclusion that the best way to get something done is to do it yourself. That is not to say that the CS Department was uncooperative, but rather that the job of the CS Department is to teach us Computer *Science*, not Computer Skills or Software Engineering. Unfortunately, teaching UNIX falls more into the Skills category than the Science category. So over the summer I planned, with a lot help from Prof. Melissa O'Neill, a course that would cover the basics of UNIX. The course was mildly successful, but only reached a small fraction of the student body. This manual is the product of the UNIX course that Marshall

Pierce and I taught during the fall of 2005.

This manual is here to make sure that students are not as lost as I was, that they have some guidance, or at least know where to look for answers. For the most part I had to teach myself and while I love the freedom that that allowed, it can also be unbearably frustrating. I hope that this book relieves you of some frustration and provides some guidance.

■ How To Read This Manual

This book is not exactly designed to be read cover to cover, although that style is certainly feasible. Instead, students should find those sections that interest them and peruse them. However, I do have one warning: *you should read Chapter One.*

If you don't read anything else in this manual please read the first chapter. It contains advice and specific pieces of information that you will need to know to effectively use your CS Department account. It also lists where you can find more information about UNIX. You can skip the other chapters if you wish, but do please read the first chapter.

Chapters Two and Three cover what you'll need to know about UNIX in adequate detail. Chapter Two details the very basics of what you should know. If you are new to UNIX you will want to try each command as you read the chapter. Chapter Three covers many of the more advanced things that you can do in the shell. If there is some sort of complex operation that you want to perform on the command line, look there first.

Chapter Four is all about what you need to know to program at Mudd. This chapter contains some much needed guidance for first time UNIX programmers, like how to write good UNIX programs. Mudd programmers, in general, should benefit from the discussion about how to properly structure, write, and comment programs. If you want a good grade in CS70 you should read the appropriate sections carefully. CS60 and CS70 students will also find that the chapter discusses some of the tools that should prove useful as they do their homework.

■ Notation

Before continuing you should be aware of the notation used in this book. I've tried to adopt the notation used in other references, so knowing this notation isn't entirely useless. The names of various programs will appear in italics *like this*. Commands to be typed into the shell that appear in the

text are typeset in bold `like this`. Most often, however, I will try to show the full command line

```
% like [this.]
Output from a command would go here.
```

The `%` symbol will serve as a basic prompt in most examples. Optional arguments or switches for a command will be shown in square braces, like the `this` "argument" shown in the example above.

UNIX has a number of different shells that users can choose from. By default all accounts created during and after the 2005-2006 school year use *zsh* as the default shell. In case your account was created before this *tcsh* was the default shell. This manual assumes that you are using *zsh* as your shell. When I refer to "the shell" in this text I am referring to *zsh*, however, most shells are extremely similar and much of what you learn here applies to the other UNIX shells.

UNIX programs use a variety of keyboard "shortcuts", much like the copy/-paste keyboard shortcuts in Windows. These key combinations will be abbreviated `C-X` for Control plus some key `X` and `M-X` for Meta (on most keyboards Alt) plus some key `X`. An alternative way to denote `C-X` is to put a caret (`^`) and then the character, such as `^d`. Note also that the case of `X` matters, and you may sometimes have to use the Shift key in addition to Control and Meta (Alt).

Chapter 1

Getting Started

1.1 Logging In

In order to log into any of the CS Department computers you need to first obtain a user account. Speak with the CS Department System Administrator, Tim Buchheim, to get an account. His office is located at Beckman B101. By having an account you agree to follow the HMC CS Department System Policy; be sure that you read and understand this document before logging in.

In either the Terminal Room or the Graphics Lab you login by entering your username and password in the login window. Note: when logging into the old Sun boxes in the Graphics Lab nothing will be echoed to the screen when you type your password. If you ever encounter problems logging in, or using the computers in any way, please see either Tim Buchheim or email `staff@cs.hmc.edu`.

Once you are logged in you will be able to access everything in your home directory. If possible try to work off the machine you've logged into by running web browsers, text editors, and other programs locally. Please note, however, that to submit your homework assignments you will need to be logged in to either `knuth` or `turing`. See section **??** for more information about the *submit* command.

Depending on which computer you log into you will be presented with either an X-Windows Sawfish session or a Mac OS X session. Both of these are graphical environments that can be customized to suit your needs. The finer features of each of these environments is out of the scope of this manual, but this manual does cover some of the basics.

1.2 Opening a Shell

Knowing how to start a shell session is the most basic UNIX task you need to do. If you have logged into one of the CS Department servers remotely then you'll already have a shell open. If, however, you have logged into one of the computers

in either the Terminal Room or the Graphics Lab, or are using your own machine, then you'll probably not have a shell open.

Just to make things a bit less confusing, let's clear up some terminology. A shell is actually a program, like Word or Safari, but differs in that its purpose is to run other programs while providing a useful and productive environment for the user. That last part wasn't always true—the first shell was really only for running programs and didn't provide much to help out users.

A terminal, or a console, is a connection through which you can talk to the computer. At one time it consisted of actual hardware—a keyboard and a screen. In today's world we almost exclusively use *emulated* terminals, but the ideas are still the same. What this all means is that the terminal is where the shell, and all the programs the shell executes, do their I/O. To get a shell you first start a terminal emulator, which will then start your shell. That shell is then connected to that terminal and will perform I/O exclusively through that terminal. Colloquially, however, opening a terminal, console, or shell are all interchangeable and mean the same thing, though technically they are different.

The following subsections list what terminal emulators are available for the specified operating system. Students should be aware that the sections on Windows and Linux apply to their own dorm machines, and not machines maintained by the Computer Science Department.

1.2.1 OS X and SunOS

OS X comes bundled with two terminal emulators: *Terminal.app* and *xterm*. A shortcut for *Terminal.app* should either be present in your dock (drag one there if you don't have it), or you can find it using the Finder in `/Applications/Utilities/Terminal.app`. To run *xterm* you'll first need to start *X11.app*. *X11.app*, when it first starts, starts an *xterm*.

Terminal.app is generally the nicer emulator to use. It offers a very wide range of features, including window transparency (useful for when you need to see the window in behind), font and color selection, and a very large scroll-back buffer. The one lacking feature is that you cannot run X-Windows based programs from *Terminal.app*.

xterm is just a standard implementation and offers no special features other than the ability to run X-Windows applications. Try to avoid running *xterm* on OS X when you can.

On the Sun boxes you should have an *xterm* window open when you log in. If not you can find it, and other terminal emulators, in the desktop right-click menu. More information about how to use and modify *xterm* to your liking can be found online.

1.2.2 Windows

The Computer Science Department does not maintain any Windows machines for general student use. Therefore, to use CS Department machines you will need to

remotely login. See section 1.3.2 for more information about how to login remotely from Windows.

1.2.3 Linux

For those students running Linux, they should be pleased to know that it offers a remarkable number of terminal emulators. The list below enumerates some of the available emulators. They are in no particular order.

- xterm
- eterm
- aterm
- rxvt
- gnome-terminal
- konsole

Students running either KDE or GNOME will probably be most comfortable using their desktops' bundled terminal emulator, *konsole* and *gnome-terminal*, respectively. These two offer a more friendly set of features to new users.

1.3 Remote Login

Every CS Department computer (servers and those in the Terminal Room and Graphics Lab) runs *sshd* and can be accessed using the Secure Shell (SSH) protocol.

For students new to UNIX, this protocol provides a secure method for remote login. This means that you could login to `knuth` from your dorm machine and have complete access to all of your data and all the resources available on `knuth` (i.e. lots of CPU cycles). Lazy students, students that prefer using their own computers, and students who live far away (hello Pomona people!) will all find this facility very useful. To learn more about SSH, how to use it, and what it offers, see section 3.1.

1.3.1 OS X and Linux

To use SSH from within OS X and Linux you will need to open a shell. See section 1.2 for instructions on how to open a shell. You will then need to execute the *ssh* program. For example, to log in to `knuth` you would execute:

```
% ssh yourusername@knuth.cs.hmc.edu
Password:
```

You would then enter your password. Nothing will appear as you type; this is a security feature. If successful you will now be logged in.

Note that I had to specify the full URL of the server to connect to, in this case `knuth.cs.hmc.edu`. All HMC CS Department servers have the suffix `.cs.hmc.edu`.

1.3.2 Windows

Windows, by default, doesn't offer a real UNIX command line. To login to the CS Department servers you'll need a program like PuTTY, which is its own terminal emulator. You can download PuTTY for free at `http://www.chiark.greenend.org.uk/~sgtatham/putty/`. The site also includes documentation on how to use PuTTY.

For a more complete reference on some of the different implementations of SSH for Windows, and other operating systems, see the Department's QREF.

1.4 Your Home Directory

Your home directory is a place of your own, where you can play, have fun and learn. Or completely trash, keep disorganized, and delete everything with no one to blame but yourself. More than anything, your home directory is your home for the next three (two/one) years, and is where you will (or should) store all of your homework for many of your CS classes.

So just what is a home directory? A home directory is a directory tree on a UNIX system that belongs entirely to you. Every normal user on the system has a home directory that they can use to store their files without fear that other users might accidently delete, alter, or even read them. While the delete and alter part should be obvious (wouldn't it be kind of bad if your friend mistyped something and deleted all of your homework?), the read part may not be so obvious. At Mudd we live under the Honor Code, and so that prohibits students from sharing or stealing other student's solutions. Unfortunately the rest of the world doesn't live under the Honor Code and so more practical solutions exist to keep people from reading your files.

Here are some tips for using your home directory:

- Your home directory is of a finite size (120MB at the moment) so use that space well. Don't store 50MB of porn there and delete cached files when you need space. To check how much of your quota you are using use the *checkQuota* command. If absolutely necessary you can have CS Staff increase the size of your quota.
- Don't let your quota fill up!
- I repeat: Don't let your quota fill up! The 120MB limit is actually a "soft" quota, meaning that you can create files up to the "hard" quota. The hard quota is at 150MB. However, if you go over the soft quota for more than a week, you won't be able to create any new files and will need to delete files until you are under quota. To see how large your files are see section 3.3.4.
- Email is an official means of communication for both the College and the CS Department. If you completely fill your CS quota, you won't be able to receive any email at your `@cs.hmc.edu` account; it will bounce. Please keep both your **CS and regular Mudd email account under quota.**

- Your home directory is backed-up on a regular basis (every 4 hours to be pedantic). If you accidently delete files and need them recovered send an email to staff@cs.hmc.edu. This process can take a few hours, so don't delete your homework at 11:55PM. We are experimenting with a system whereby students will be able to recover their files themselves. See Tim Buchheim for more information.
- Keep your files organized. By default every account has a `courses` directory, with sub-directories for many CS classes. Try to keep your homework organized in the appropriate directories. This will make your life much, much easier when you go looking for that "one homework you did in CS60" two years from now.
- Make new directories liberally. Ideally your home directory should be nothing but subdirectories containing your homework, projects, scripts, downloads, documents, trash, and anything else you may create/generate/etc. Having lots of directories should also help you keep organized.
- Get outside your home directory and browse the neighborhood! Just by reading a few random files in the UNIX directory tree you can learn a lot about how UNIX systems are organized and how a great many programs are configured. Directories to check out are `/etc`, `/usr/bin`, and `/var`

1.5 Changing Your Password

After logging in for the very first time the first thing you should do is change your password. Even though the password you have been given has been randomly generated, it is not secure—you need to be the only one to know your password! Never tell any other person your CS Department, or any other, password for any reason. Your password MUST conform to the following specifications:

- be at least 8 characters in length;
- have at least 2 upper case characters;
- have at least 2 non-alphanumeric characters (@ ! # $ % & . , ( ) and spaces);
- and have at least 1 digit.

You should never use a plain dictionary word as a password, however, it does provide a good start. Say we start with the word 'elephant'. This is clearly not a good password, since it's a plain dictionary word, but by making a few simple changes can be made into an acceptable password. First we change the 'l' to a 1, next make the 'h' into a # and the 'a' into a @, then capitalize the first and last letters. This produces the password `E1ep#@nT`. While this may not be an incredibly strong password, something generated by a known random source is best, for most purposes it would be fine.

To change your password execute the *passwd* command at the shell prompt. Follow the directions on screen to change your password. If you receive an error be sure to talk to the System Administrator, or email `staff@cs.hmc.edu`.

1.6 Finding Help

The goal of this manual is not to teach you UNIX, but to give you enough of the basics so that you'll be able to teach yourself. However, sometimes you have to ask for help, and often times this is the best way to learn something new. Don't beat your head against the wall if someone you know already has the answer and can help you figure it out!

1.6.1 CS Department

The CS Department employs Consultants, students whose job it is to help you out with questions you may have about UNIX, your account, or even homework (Grutors generally handle homework). Consultants maintain hours (found at `www.cs.hmc.edu`) in the Terminal Room and will usually have a sign reading "CONSULTANT" sitting on the top of their monitor. Please ask them questions and make them work for their pay!

The CS Department also employs Staff, students whose job is to maintain the various servers and labs that the Department owns. You'll most often find Staff members sitting in the Machine Room, which houses our servers and sits at the bottom of the stairs past the two labs, rather than in the Terminal Room. Email is the best way to contact staff if you have problems: staff@cs.hmc.edu.

Complementing the content of this manual are the QREFs that the HMC CS Department maintains at `www.cs.hmc.edu/qref/`. This is documentation written with you in mind and covers a variety of topics outside the range of this manual. Not just that, but almost all apply directly to our systems, so they will usually be both useful and correct. From time to time I will refer to those QREFs that have more information about a particular topic. Please use this resource whenever you can.

1.6.2 Online

The one resource that seems know to everything, and the first one you should always ask, is Google. Most likely someone else has already asked your question or taken the time to solve and write up whatever solution you are looking for. Besides random message boards and forums, Google can lead you to many other very good web-based resources dedicated to answering the questions new users have. Here are three that you may want to check out:

- `www.linuxquestions.org`;
- `www.linuxdoc.org`; and
- `www.freebsd.org/handbook`.

The first links to a newb friendly Linux forum. The second contains a massive amount of Linux documentation. One particularly good resource is the FreeBSD Handbook, which is the last link in the list. It includes almost all of the information you would ever need to run and maintain a box running FreeBSD. Most of the information is sufficiently general to be of use to new users.

1.6.3 Offline

Each of the above resources are excellent, but there are times when you don't have access to the web or (god forbid) are away from Mudd. In these times you can thank the heavens that UNIX includes much of its own documentation in the form of manpages, info documents, and the files in `/usr/share/doc`. See section 2.6 for more information about each of the listed documentation forms.

- Manpages are myriad, but terse, documents that are usually very helpful if you are able to read them.
- Info (command `info`) is a very comprehensive documentation system, but is as a rule poorly organized and difficult to use. It contains documentation for most GNU utilities, like *ls*, *tar*, and *emacs*.
- Lastly, most programs when installed install both a manpage and some more comprehensive documentation in `/usr/share/doc/`. This documentation should be more than enough to get you started using any new programs you may encounter.

1.6.4 Print

If you are at all interested in learning about UNIX, or want to take any CS classes beyond CS60, then you really ought to buy a few books. O'Reilly publishes many excellent books covering various aspects of UNIX. One book in particular, *UNIX Power Tools*, should be purchased. A thorough understanding of everything in *UNIX Power Tools* will indeed make you a Power User. Here are some other titles to guide you:

- *The C Programming Language (2nd Edition)* (aka. K&R);
- *The Art of UNIX Programming*; and
- *From BASH to Z Shell*.

You can find all of these books, and many more, in the CS Department library in the Terminal Room. Note: these books cannot be checked out, or leave the Terminal Room. We're sorry for the inconvenience, but the real library is just *upstairs*.

1.7 `request`ing Help

In order to keep track of those tasks that need to be taken care of in the daily maintenance of the CS Department system, Staff makes use of a program called *request*. This program allows Staff, Faculty, and Students (that means you!) to create and open "requests". For example, if you needed a particular program

installed for research work, or needed to be added to a particular UNIX group, then you would open a request.

When creating requests you ought to be aware of a few things.

1. It is a request, and it may not be completed in any reasonable amount of time. If something is urgent then you should say so, but no one is under any obligation to fulfill your every whim.
2. Please do not use *request* for what might be considered trivial items. This might include typos or spelling errors in Department web pages. Non-trivial things include **anything** that might keep you from getting your work done, and you should not hesitate to open a request immediately in such cases.
3. Do not be rude. These are your fellow classmates who are helping you out. Show them the same respect you would in the classroom or out.
4. *request* is installed only on `knuth` and `turing`. We are sorry for the inconvenience.

To create a new request from the command line use the `-n` switch, like so:

```
% request -n
(Lots of output here)
```

request has many switches, all of which you can see by using the `--help` switch.

1.8 Computing Facilities

Their are many different computing facilities on-campus. Discussed in the next two sections are the facilities offered by the CS Department and those offered by CIS, the college's computer department. Note that Engineering and Math also maintain computing resources, but are not discussed here.

1.8.1 Computer Science Department

The CS Department maintains two general use computer labs in the Libra complex, Beckman B105 and B102. The combo to the rooms can be obtained from the CS Department System Administrator. Beckman B102 is informally known as the Terminal Room, and is where you'll likely do most of your work at first. The lab is stocked with a total of 24 Mac Minis, and all are available for student use. A printer (gute.cs.hmc.edu) is also available. Beckman B105 is informally known as the Graphics Lab. It is scheduled to be restocked with a newer set of computers, but currently contains Ultra Sparc II machines running Solaris 9. But don't despair, at the moment there are two PowerMacs in the Graphics Lab.

The CS Department also maintains a handful of shell servers that students can log into remotely. These servers are where you will do (or should do) the majority of your work for most CS classes, especially CS60 and CS70. The server that you'll find yourself working on the most is `knuth`. `knuth` is an IBM server with 4 Pentium 4 Xeon processors each running at 3.6Ghz, 8GB of RAM, 36GB RAID 1,

and is being powered by Debian Linux. Other general use servers include `turing` and `wilkes`. `turing` is the old UltraSparc CS Department server and has served us well for many years. `wilkes` is a dual-processor Dell server, also running Debian Linux, and is mainly used by students in CS105. Below is a full listing of our servers. A given server can be reached at at `SERVER.cs.hmc.edu`.

- `knuth`: recommended shell server (Debian Linux)
- `turing`: old shell server (Solaris 9)
- `wilkes`: alternate shell server (Debian Linux)
- `muddcs`: web server and DNS server
- `ark`: NFS file server
- `durandal`: LDAP
- `cortana`: faculty/student SAMBA server

1.8.2 CIS

CIS is a separate department that administers the student computer labs in downstairs Parsons, the lab in the LAC, as well as the printers and scanners at those locations. CIS is responsible for maintaining the student network and is also responsible for on-campus email delivery (i.e. any mail arriving at your `@hmc.edu` acount). Please direct all questions regarding CIS resources (computers, printers, etc.) to the CIS office in downstairs Parsons.

1.9 X-Windows

X is the system by which UNIX programs are able to have a graphical user interface, or GUI. X handles the rendering of windows, and keyboard and mouse events. Because of the client/server architecture of X, where the processing occurs, and where the windows are drawn, mice clicked, and keys punched, does not much matter. Because of this, you can login to a server and run a graphical program from that server as long as the correct settings are enabled. For example, you could run Firefox on `knuth` (knuth would literally be running the firefox process) and its window would be displayed on your machine.

To run an X enabled program, like Firefox, an X server must be enabled. On most of our shell servers (`knuth`, etc.) we do **not** run X because of the heavy load it puts on the server and because most graphical programs can be run locally. You can run X elsewhere, however. By default you are already running X when you log in to the Sun boxes in the Graphics Lab. On the Macs you'll need to start *X11.app* to have X services. There are also X servers available for Windows.

The focus of this manual, however, is on the command line. However, you'll most often be running a shell from within X rather than logging in directory to a console. Therefore, you should probably spend a little time familiarizing yourself with X.

Chapter 2

Shell Basics

2.1 Introduction

You might think that using the command line is very 1984, with DOS, and your parent's bad taste in music and hairstyles, but the command line is actually a very, very powerful tool. The shell allows you to quickly, easily, and efficiently process anything that can be easily represented in a textual format. But, like any tool, you have to learn to properly use it first. Fortunately, the shell is fairly simple. You type in a command, or a series of commands, along with some arguments, and the shell executes the command(s). If this sounds a bit like programming, it ought to: UNIX was made by and for programmers. As a result, the UNIX environment is well suited to programming and should give you a new way to think about human/computer interaction.

Before you begin using the shell, there are some things you should know about UNIX. UNIX is a multiuser and multitasking environment. This means that UNIX supports many users being logged in at the same time and that each user can run multiple programs without interfering with other programs that other users may be running. This should seem familiar to users of Windows and OS X.

UNIX filesystems are case sensitive. This means that if you have a file named `bunny` it is not the same as either `Bunny` or `BUNNY`. This too should seem familiar to users of Windows, but OS X users will want to note that most OS X filesystems are *not* case sensitive. Instead, the HFS+ filesystem on OS X is case preserving, so a user could create a file named `Rabbit` and delete it using the name `rABBIT`. (Apple does now offer the option of making HFS+ case sensitive. It is not recommended that this mode be used as it can break some programs.)

UNIX is old, nearly 40 years in fact. As a result of this long history there are many anachronisms that may seem out-of-place, just silly, or really annoying. In any case, UNIX has proven itself and is unlikely to change much, or go away, in the next 40 years.

2.2 Command Syntax

The shell understands two basic types of commands: program names and built-in commands. The distinction is very simple: when you type a program name and execute that as a command that program is executed; when you type a built-in command and execute it the shell you are running handles the command. In fact, you've already run a separate program when you changed your password. When you ran the `passwd` command the *passwd* program was executed by the shell. It ran, took some input, changed your password, and exited when done returning you to the shell. The syntax of either type of command is the same. The syntax of a command is very simple and looks like this

```
% command [switches] [argument_1] ... [argument_n]
```

As you can see, all switches and arguments on the command line are separated by spaces. This important—losing a space drastically alters the meaning of what you've typed. Therefore, watch what you type since you can easily mistype and end up with something very bad.

You can supply multiple commands at once, they just need to be separated by a semi-colon, like this:

```
% command_1 ; command_2 ; command_3 ; ... ; command_n
```

The commands will be executed in the order given and there will be no breaks between the output of each command.

Switches are supplied to alter the default behavior of a command. You might also see them referred to as either flags or options. Switches usually follow one of two different styles. The UNIX style is preceded by a single dash and followed by an alpha-numeric letter; some commands have so many switches that uppercase letters and digits are used. Shown below is the default output of the program *ls* (*ls* lists all the files and directories in the current working directory):

```
% ls
pr0n.jpg        secrets.txt     unixbook.pdf
```

Below is an example of *ls* using a UNIX style `-l` switch on the same directory:

```
% ls -l
-rw-r--r--   1 mkegel  mkegel  143288 Oct 30  2003 pr0n.jpg
-rw-r--r--   1 root    wheel   98455  Oct 30  2003 secrets.txt
-rw-r--r--   1 mkegel  mkegel  390997 Oct 30  2003 unixbook.pdf
```

As you can see, the `-l` switch causes *ls* to output a more complete description of each file in the directory. Multiple UNIX style switches can usually be combined, like so:

```
% ls -lS
-rw-r--r--      1 mkegel  mkegel   34493 Dec 11 04:36 sf20020213.gif
-rw-r--r--      1 mkegel  mkegel   25129 Jun 27 00:24 scrippsieangel.png
-rw-r--r--      1 mkegel  mkegel    4235 Dec 10 16:33 passwords
```

The other meanings of the other columns will explained in time, but the fifth column indicates the size of the file. The `-S` switch causes *ls* to sort by size in decreasing order.

The other style of switches is the GNU style. GNU style switches use two dashes followed by a whole word. For example, the GNU program *tar* (used to create tape archives) takes the `--create` switch, which creates an archive of the specified files. Users should note that *tar*, like many other programs, may take either a GNU or a UNIX style switch for the same action. In *tar* the `-c` switch has the same functionality as `--create`.

The arguments a command receives are extremely important. Commands don't usually exist naked on the command line—they usually need data to operate on and arguments provide a means to furnish this data. Arguments can represent anything: directory names, filenames, email addresses, names of other commands, etc.

Spacing is very important when typing on the command line. In order to be separate, arguments and switches must be separated by one or more spaces. The simple program *echo* demonstrates how to pass multiple arguments to a program. You can probably guess what *echo* does:

```
% echo Merry Christmas and a    Happy    New    Year
Merry Christmas and a Happy New Year
```

There are times when you will need to execute a command that is in the current working directory, but that directory is not in your `PATH` (see section 3.7.3 for more information). To execute such a command precede it with the string `./`, like this:

```
% ./command_in_this_directory
```

> Programmer's Note
> The shell makes no distinction between switches and arguments. To the shell, everything typed after the command is all the same. It is the command itself that enforces the distinction between switches and arguments. When a program executes and receives arguments from the command line it receives a list of strings (`argv`). This is important! The whole command line is not passed in as one long string. Instead, the command line is parsed such that words separated by spaces are broken up into separate strings and passed into the program as a list. This makes argument parsing easier, but libraries, like getopt, have been written specifically to do this parsing for you. You will get to work with switches and arguments in CS70.

2.3 Basic Commands

Like Chinese characters, UNIX commands make absolutely no sense the first time you see them, but after a while it all becomes second nature. Each section below covers the most basic UNIX commands; those commands that you'll need to know by heart. Don't be discouraged if you don't memorize them and each of their switches on your first try. Just keep at it.

2.3.1 Taking Inventory With `ls`

The *ls* programs prints out the files in your current directory. However, by default *ls* will only print out those files (or directories) that DO NOT begin with a '.' or dot. These files, known as dot-files, generally contain configuration information for the programs you run, like *emacs* or *jedit*, and are not usually to be tampered with. Here is an example of using *ls*:

```
% ls
file_one
got_another
last_file
```

Some of the more common options to *ls* are listed below. If more than one switch is listed, the one on the left refers to the OS X version, the right one to the GNU/Linux version.

Synopsis:
`ls [-alFG] [file ...]`
Switches:

`-a`	ALL files in the directory are listed
`-l`	long format; lists extra information about each file
`-F`	pretty output; directories have a '/' appended, etc
`-G | --color`	enables colorized output; very useful

2.3.2 Navigating Directories With `pwd` and `cd`

The *pwd* command prints what your current working directory (cwd) is at the moment. You can think of the UNIX directory hierarchy as a tree and your current working directory as a single node in that tree. You will generally execute programs or specify pathnames according to your cwd. The following example shows how to print your pwd. Note that the default prompt displays your cwd.

```
% pwd
/home/your\_user\_name
```

The *cd* command is built into your shell and changes your current working directory. You use *cd* to navigate the UNIX directory hierarchy from the command line. When given no arguments *cd* will take you to your home directory. The example below shows how to use *cd* to change to a directory:

```
% cd your_directory_here
```

Let's assume for a moment that you have just logged in and need to work on your CS60 assignment. Your account, by default, has a `courses` directory, with sub-directories for many classes in the CS department. To have *cd* change your cwd to `~/courses/cs60` you would type the following when you log in:

```
% cd courses/cs60
```

Though you won't see any output, you should see your prompt change to indicate that your cwd is now ~/courses/cs60. You could also get to the same directory by typing:

```
% cd /home/your_user_name/courses/cs60
```

These two example illustrate the difference between relative paths and absolute paths. The first is an example of using relative paths, the second uses an absolute path. The only difference is that absolute paths start with a /, that is they start at the root. However, you'll most often use relative paths. Relative paths are specified according to the cwd. In the example above, if your current directory hadn't been /home/your_user_name then its very unlikely that *cd* would have succeeded.

Whenever you see a ~ at the start of a pathname, that's just the shell being helpful. As you may have guessed, it's just an abbreviation for the path to your home directory: /home/your_user_name.

There are two very important file paths to know: '.' and '..'. Because the UNIX hierarchy is a tree, you often need to refer to the parent of the current directory. Typing `cd ..` will take you to the parent directory of the cwd. The '.' path refers to the cwd, which you sometimes need to refer to for a command. You'll sometimes see these used in pathnames. Note: these files are present in *every* directory. Even directories that are "empty" will still have these two files.

> **UNIX Filesystem**
> One of the ideas that UNIX borrowed from its intellectual predecessor, MULTICS, was the idea of a unified namespace. Unlike Windows where your hard drive starts at C:, the floppy disk is A:, and the CDROM is D: or E:, UNIX starts at /, or root. You can visualize the UNIX directory hierarchy as a tree, with / as the root of the tree, hence the name. All devices, all partitions, all files, all directories, everything exists as a file somewhere on the tree. Your home directory, for instance, exists as a directory at /home/your_user_name. The idea that "everything is file" is very powerful and is fundamental to UNIX, about which you can read more later.

2.3.3 Managing Files with `mv`, `cp`, and `rm`

The three commands, `mv`, `cp`, and `rm`, are all that you'll need to manage files and stand for move, copy, and remove, respectively. Both *mv* and *cp* are similar in function and mostly follow the same semantic rules. That is, if the last argument is a directory then the file(s) listed are moved/copied to that directory. A special case exists when there are two arguments and the second is not a directory. In this case the file is renamed/copied with this new filename. An error is generated if you supply only one argument. To copy the contents of a directory you'll need to use the `-r` switch to recursively copy through the directory hierarchy. Some examples follow:

Moving the files `foo`, `bar`, `baz`, and `bot` to the directory `dir`:

```
$ mv foo bar baz bot dir/
```

Copying the contents of directory `millerlite` to directory `litebeers`:

```
$ cp -r millerlite litebeers
```

The easiest command to shoot yourself in the foot with is *rm*. Not only does it delete files, there is generally no way to recover them! Luckily for you, the CS Department maintains backups of everything on the system, so if you do delete a file it can usually be recovered. We are currently running backups every four hours, so any files created and deleted within this window may be permanently lost. On systems without backups, deletion is usually permanent. The example below shows how to delete files `fred`, `bob`, and `sponge`:

```
$ rm fred bob sponge
```

To delete an entire directory tree you need to use the `-r` switch, like so:

```
$ rm -r some_directory_here/
```

This switch causes *rm* to recurse down the directory tree. If you try to delete a directory without this switch you'll receive an error.

One important thing to remember is that you can only delete files that belong to you. See section 2.3.7 for more information about file ownership and changing permissions.

> **What ought the command `mv a b c` do?**
> Think about this a minute...ready? Fail, when `c` isn't a directory, is the correct answer. But to keep you on your toes early versions of *mv* would actually rename `a` to `c`, then `b` to `c`, thus erasing the contents of `a`. Newer versions of *mv* check to make sure that the last argument is a directory, before proceeding, when there are three or more arguments. This is just another reminder to type what you mean, and mean what you type!

2.3.4 Managing Directories with `mkdir` and `rmdir`

You can use the *mkdir* command to create a single directory, or all the directories in a path if you so choose. The example below shows how to create the directories `aSimpleDirectory` and `anotherDirectory`:

```
$ mkdir aSimpleDirectory anotherDirectory
```

To have *mkdir* create intermediate directories in a given path you must supply the `-p` switch, like so:

```
$ mkdir -p some/really/large/directory/tree
```

The *rmdir* program deletes directories, but `rm -r` is generally much more useful since *rmdir* will only delete a directory that is empty. You can delete an empty directory like so:

```
$ rmdir anEmptyDirectory
```

2.3.5 `clear`ing the screen

The `clear` command (alternately `C-l`) clears the screen and is useful for when your screen has garbage all over. The command actually works by scrolling down until the only thing showing is your prompt and so doesn't actually delete anything from your terminal. This means you can still access what was on your screen by scrolling up, either with the scroll-wheel on the mouse or by using the pageup key.

2.3.6 Viewing Files With `less`, `more`, and `most`

Plain ASCII text is the native language of the command line and most of the files you'll create will be plain text. Source code, latex documents, configuration files, and numerous other files all use plain text. Since these files are so common, you'll need a convenient way to view them.

The easiest and quickest way to *view* plain text files is with a pager. You can't edit files with a pager—you need an editor for that. Functionally, a pager allows a user to "page" through files one screen at a time. The first pager, *more*, is a very simple program. Using the spacebar you can page through a file . However, once you page down you can't go back. This makes *more* the least useful pager, but since it is the default for some other programs, it will always be available on any UNIX system and you may have to use it.

A much better pager to use than *more* is *less*. Logically following from the old adage of "less is more", *less* is more than *more*. *less* has some very nice features: you can scroll back and forth in a file and search for text.

Better than either *less* or *more* is *most*. We only have *most* installed on the Linux servers, and it does have some nifty features (ooh! color). Try it if you ever get a chance.

Files to view are passed in as arguments to the pager of your choice. Below is an example of how to view `someRandomFile.txt` with the *less* pager.

```
$ less someRandomFile.txt
```

Once you have the file open it helps to know what the different controls are. Here are some of the different controls for *less*:

up/down	move up/down one line
RETURN	move down one line
left/right	move half a screen width left/right
f/spacebar	move forward by one screen
b	move backward by one screen
g	move to the beginning of the text
G	move to the end of the text
/pattern	press /, then type a *pattern* to search for, then hit return
n	search for the next instance of *pattern*
N	search for the previous instance of *pattern*
h	help menu
q	quit

2.3.7 Changing File Permissions with `chmod`, `chgrp`, `chown`

UNIX uses a very basic (now arguably insufficient) security model. Everything on the system is owned by some user: files, devices, processes, everything. We will just concern ourselves with files for now.

A file has an owner and a group membership. The owner can be any user on the system. Every user on the system also has a default group that they are a member of. For example, normally when you create a new file you will be the owner and the file will be belong to your default group.

Groups are a bit more complicated. Groups are sets of users, but are severely limited in that they cannot contain other groups. In effect, they allow for easier management of the system by giving a well-defined set of users special privileges. For example, we add students who are members of CS Staff to the `staff` group. On our systems, normal users, like you, are members of the `students` group.

One user that every UNIX user should be aware of is `root`. This superuser can do anything on the system. The account can be used to read/write/delete any file on the system, kill any processes, and generally wreak havoc in the wrong hands. In effect, `root` owns and maintains the system. This is why getting hacked is generally seen as a bad thing. As a normal user you don't, and shouldn't, have the kinds of powers `root` has over the system, since it makes both the system and your data more secure.

To view the owner and group of a file use *ls* with the `-l` switch, like this:

```
% ls -l
-rw-r--r--   1 mkegel  staff   143288 Oct 30  2003 pr0n.jpg
-r-xr-xr-x   1 mkegel  staff   143288 Oct 30  2003 script.sh
-rw-rw-rw-   1 root    wheel   98455  Oct 30  2003 secrets.txt
-rw-r--r--   1 root    mkegel  390997 Oct 30  2003 unixbook.pdf
```

To change a file's owner you would use the *chown* command. For example, I could change the owner of `unixbook.pdf` from `root` to `mkegel`:

```
% chown mkegel unixbook.pdf
```

Note: you can only change a file's owner if you are the superuser (i.e. root), or have superuser privileges.

To change a file's group you use the *chgrp* command. For example, I could change the group of `pr0n.jpg` from `mkegel` to `students`:

```
% chgrp students pr0n.jpg
```

Note: group changes can only be performed by the file's owner, and that user must be a member of the group that the file's group ownership is being changed to.

Associated with a file's owner and group are a set of ten permission bits that define whether you can read/write/execute (rwx) the file. Directories have the same permissions but they mean something a bit different and stand for list/create/access files in the directory. In the output of *ls* above the first column shows

these bits. The first bit is unimportant for most students. The next three represent the permissions for the file's owner; the next three for the file's group; the next three for the other users on the the system. From the output above you can see that everyone can both read and write the file owned by `root` (a bad thing) and that everyone can read and execute `script.sh` (also a bad thing).

A succinct way of describing permissions is with octal notation. Octal notation works by assigning the values 0 to 7 to a group of three bits. In this case you need three octal digits: one for owner, one for group, and one for others. The octal representations of the above files are: 644, 555, 666, and 644. This diagram should help clarify the situation:

```
owner|group|others
 rwx   rwx   rwx
 421   421   421
```

To change the permissions of a file you use the *chmod* command. You can use either octal permissions or a longer text format. The text format specifies:

Who:		What:		Permissions:	
u	user (owner)	-	subtract permissions	r	read/list
g	group	+	add permissions	w	write/create
o	others	=	set exact permissions	x	execute/access
a	all				

For example, assuming I was root I could change the permissions of `secrets.jpg` to have only `root` read/write access with any of the following commands:

```
% chmod 600       secrets.jpg
% chmod g-rw,o-rw secrets.jpg
% chmod go-rw     secrets.jpg
```

The last thing you need to know about permissions is your `umask`. By default when you create a new file all the permissions are turned ON, that is a file would have permissions 777. Your `umask` controls which permissions are then immediately turned OFF. For example if your `umask` was something reasonable like 076, then no one in your group or otherwise could read or write your files, and files would end up being 701. World execute is needed because of apache; see section 3.11 for more information. Your `umask` is by default set to a reasonable value, but if you would like to change it, you can find it's entry in your `.zshrc`. See section 3.7 for more information about customizing your shell and your `.zshrc` file.

2.3.8 Reading Email with `pine` and `mutt`

You will receive email at your CS account that you will want to read, or delete depending on its contents. To read your email you can use either *pine* or *mutt*. Both are suitable email readers, though *mutt* does have more features (it was also

created by a Mudder ☺). You can find documentation about both programs and much more online through the CS Department QREFs.

> Forwarding Email
> If you have a different account that you use more frequently you may want to forward your email. To forward your email, create a file named `.forward` in your home directory with the contents being the email address to forward to. You can keep a local copy of the message by prefixing your username with a backslash. You can forward to multiple addresses by separating them with a comma. For example, I forward all my email to Gmail since I want to be able to send and quickly access my email from anywhere. I also keep a local copy, in case mail gets lost. My `.forward` contains the following: `\mkegel, mark.kegel@gmail.com`

2.4 Exiting and Suspending Processes

Every program is different, but you can exit most processes by typing `C-c`. This sends a special signal to the process that by default tells programs to quit; some programs choose to ignore or reinterpret this signal. Another useful key combination that quits some processes is `C-d`, which sends the End Of File (EOF) character.

The currently running process can be suspended by typing `C-z`. This causes the process to halt execution, effectively pausing the program, and making your shell prompt to appear. New users may incorrectly think that the program has exited or been killed, when in fact it is suspended. If this happens use the *fg* command to restart the process. See section 2.5.4 for more information.

2.5 Shell Magic

The shell does much, much more than just execute commands. The shell is a program like any other, and like most programs has a number of features that makes your life easier (and some features you'll never use). Some of the more useful (maybe essential) features of the shell are outlined in the next few pages.

2.5.1 File Globs (aka Wildcards)

So far all the examples you've seen have been fairly simple. Each file or directory has been specified by name, but there are times when you want to refer to some well defined subset of all files in a directory, like all the pictures or movies. Generally, you can think of a file glob as a pattern against which filenames are matched. You define a file glob on the command line using special characters, which the shell then processes before executing the command. A wildcard is usually just a single, special character that has some meaning in a file glob.

The simplest file glob you can match against is a string. The only thing that will correctly match against the string is the string itself. For example, if you used

the glob `bob` to find a file, only those files named `bob` would match. Most of the time this is not quite the behavior you want. Instead you want one part of a file name to match a specific string, with no, or only a few, constraints other parts. The wonderful thing about file globs is that they can be combined to get behavior exactly like this.

As a simple example, say you want to backup everything in a directory. You would probably type something like this:

```
$ cp -r * .backup/
```

The * wildcard is the most basic wildcard, and matches every file or directory name in the current directory except those starting with a dot (.). So if the directory had files `bud, weis, & er` they would all match and be copied. The `.backup` directory would not match the pattern.

Rather than copy every file in the directory, you would prefer to backup only your source and header files. You might type something like this:

```
$ cp *.cc *.h .backup/
```

By combining the two expressions we get a more useful file glob, one that matches those files with a specific ending. File globs can be combined in any number and in any order, though most often a simple `*.something` will suffice.

New users should note that the . exception to the * pattern is very important. Think about what would have happened if * did actually match *every* file in a given directory in the example above. Would the results be good or bad?

Recall that in every directory there are always at least two files: . and .. If * matched everything then it would also match these two files, meaning that files in the next directory up would also be copied into `.backup`. This is probably not very good behavior.

On the other hand, this is one of those instances where UNIX, in my opinion, got it wrong. Rather than just hard code in that * does not match either . or .. the original designers thought it better to just exclude everything that began with a dot (.). Remember also that by default *ls* does not print files/directories that begin with dot (.). Both are historical artifacts which we'll have to live with forever.

Below is a listing of some other wildcards and their meanings:

?	match a single alpha-numeric character
[a-z]	match a single character in the given character class
(x|y|z)	match either x or y or z
{a,b,c}	expands to "a b c"

By combining these wildcards you can get many useful file globs. The * and ? wildcards are by far the most useful and will suffice for 90% of what you'll likely need to do. Here are some illustrative file globs and some example files that would match:

```
% ls a[a-z]*
```

```
al              alpha               azRAEL
% ls *.jpg
aliens.jpg      men_in_black.jpg    z3d.jpg
% ls images?.bmp
images0.bmp     images1.bmp         images2.bmp
% ls (red|black|green).*
black.          green.eggsnham      red.eye
```

One thing that you should always remember when using file globs is that they are provided by the shell. There are many different shells and they do not all provide the same file globbing capabilities or use the same wildcards. Your default shell, however, is more than sufficient and the wildcards shown will work as documented.

2.5.2 Special Characters

By default shells treat many characters as having a special meaning. For example, as you saw in the previous section the * character is treated as a wildcard. However, there are times when you may have a file or directory that has a special character is its name. To type a special character as a literal character you have to escape it with the \ character. For example to type a literal * on the command line it would look like this:

```
% echo \*
*
```

Notice that instead of *echo* printing all the files in ./ it printed a plain *. Note: if \ appears at the end of a line it removes the newline character, causing the command to be continued onto the next line. The following are escapable special characters:

```
# $ * ? [ ] ( ) = \ | ^ ; < > $ " ‘ ’
```

One non-printing special character is a literal space character. You escape it like any other character, inserting a space after the \, as this example demonstrates:

```
% ls file\ with\ several\ spaces
file with several spaces
```

Another way to type filenames with special characters is to surround the filename in either single quotes (') or double quotes ("). Surrounding a text string in single quotes disables all special characters; surrounding a text string in double quotes disables all but $ ’ \. So the previous example could have been:

```
% ls "file with several spaces \
> and continued to the next line"
file with several spaces and continued to the next line
```

2.5.3 Tab Completion

You may have noticed a pattern in UNIX command names—they are all short, succinct and easy to type. However, your own directories and files should have descriptive names. But descriptive names are often long and after a while become a pain to type, especially if you have to type them over and over again. An abbreviation can even be a worse idea as you may forget what it originally meant.

One of the simplest and most brilliant ideas that UNIX shells have implemented is tab completion. Say you have a really long directory name: `homework_and_other_documents`. The name of this directory has to be unique by UNIX filesystem conventions. Not only this, but the name of the directory is likely to be unique after the first few characters. That is, `homework_and_other_documents` could have a unique name after `home`. Therefore, if all you typed at the command line was `some_cmd home` you would expect the shell to be able to fill in the rest of the name for you. This is exactly what the shell will attempt to do, and succeed in the example, when you press the tab key. Note: zsh (the default shell) will show a list of possible completions if what you've typed is not a unique identifier.

Tab completion is best illustrated by trying it for yourself. Try the following experiment on the command line. Type `asp`, then hit tab. What program do you get and what does it do?

The default shell is able to tab complete many different things, including: commands, filenames, directories, PIDs, manpages, switches, etc. Nearly everything that you type on the command line can be tab completed. Beware, though, tab completion is habit forming and should be used with caution. Side effects include attempting to tab complete conversations, homework assignments, and essays.

2.5.4 Job Control

Most students will be familiar with the graphical environments of Windows and OS X. In these environments you generally run many different programs at once: a web-browser, word processor, maybe even a terminal. Even on the command line you sometimes need to be able to run multiple programs. While it may not seem like it, the shell can easily let you run and manage several different programs. This is known as job control.

The shell maintains three job (a process, or a set of processes pipelined together) groups: those running in the foreground, those running in the background, and those that are suspended. Foreground processes are able to receive keyboard and mouse input, while background processes are not. The first two groups are able to write to the terminal.

To execute a command and immediately have it run in the background you type a `&` after the command. For example, if you were running in X-Windows you could run emacs in the background, and still have your terminal by typing:

```
% emacs &
```

Knowing how to background processes like this is very useful in the X11 graphical environment and is worth remembering.

To manage jobs the shell offers two builtin commands: *fg* and *bg*. The *fg* command brings jobs into the foreground, while *bg* puts jobs in the background. In general, you have one job in the foreground and one or more jobs in the background.

If you have multiple jobs in the background, then you can access this list with the *jobs* command. Below is some sample output of *jobs*:

```
% jobs
[1]   Running                 firefox &
[2]-  Stopped                 emacs
[3]+  Stopped                 lynx http://www.google.com
```

The first column shows the job numbers. In general, whenever you use *fg* or *bg* you will need the job number. The plus and minus signs in the first column indicate the current job and previous job (job that used to be the current job), respectively. The second column indicates whether the job is running or stopped (suspended). The last column shows what command was executed.

By default, *fg* and *bg* act on the current background job. If you have only one program in the background, this will always be the current job. To refer to an arbitrary job you would type `%X`, where X is the job number. For example, given our previous jobs list, we could bring *emacs* into the foreground like so:

```
% fg %2
```

Note that it doesn't make sense to run an interactive program, like *emacs*, in the background unless you are running in a graphical environment. If you try to run *emacs* from a straight console it will stop itself.

Avoid the Write/Compile Cycle

Most new UNIX programmers unfamiliar with job control will find themselves in the write...compile...write...compile cycle. Quitting their editor, executing their make script, and then restarting their editor to fix bugs or correct errors. You can avoid this cycle by suspending your editor with C-z, executing your make script, and then restarting your editor with *fg*. This keeps your place in the editor and your settings, which can take a significant amount of time to reset if you restart your editor every time. Graphical environments don't suffer from this problem since you can keep your editor and terminal windows both open.

2.5.5 Command Line Editing

In general, you can edit your commands before you have the shell execute them. To move the cursor you can use the left and right arrow keys. The up and down arrow keys will let you cycle through previous commands you have used. All commands given are relative to the position of the "blinking cursor of doom".

C-?	delete back one character
C-d	delete current character
C-u	delete entire line
C-k	delete from the cursor forward to the newline
C-l	clears the terminal; same as using `clear`
C-y	paste the character(s) you have deleted
C-a	move the cursor to the beginning of the line
C-e	move to the end of a line
M-b	moves back a word
M-f	forward a word
C-_	undo the last thing typed

2.6 Command Documentation

One of the wonderful things about UNIX is that it includes much of its own documentation. Most operating systems come with some sort of help feature/menu/-manual, but UNIX is absolutely packed with information designed to teach you about the system and each of the different commands. And if that isn't enough, you can always read the source code and figure out *exactly* how things *really* work in UNIX.

Most UNIX systems come with at least two distinct forms of documentation: manpages and info documents. Linux programs generally include a third in the form of html pages installed to `/usr/share/doc`. Each section below briefly discusses each of these three types of documentation.

2.6.1 Manpages

So this is all very cool, but exactly how do you find commands (and what they do, and what arguments they take, etc) in the first place? Originally, UNIX systems shipped with a bound programmer's manual. The dead tree version has thankfully been replaced by online documentation in the form of manpages. The manpage for a given command contains a full explanation of what that command does, how its arguments are formatted on the command line, what switches the command takes, etc. Examples of actually using the command are usually absent and can be frustrating for new users. An example of using *man* is shown below:

```
% man ls
```

This would display the manpage for the *ls* program. Each manpage is formatted roughly the same way and the *ls* manpage is a representative example. You should browse through it now. By default a manpage is displayed using the *more* pager. Since this is less than optimal, we've setup our default accounts to use *less* as the pager for *man*. You can use the controls listed above for *less* when reading a manpage.

Manpages cover more than just system commands. As a programmer you need to know about more than just which commands to use. You want to know about system calls, library routines, file formats, etc. To make life easier manpages are organized into sections. The different sections and what they cover are enumerated below:

1	Commands available to users
2	UNIX and C system calls
3	C library routines
4	Special file names (Devices and Device Drivers)
5	File formats, protocols, and conventions for files used by Unix
6	Games
7	Conventions, Macro packages, Word processing packages and Misc.
8	System administration commands and procedures

The *man* program actually has its own documentation stored as a manpage, which you can access by typing:

```
% man man
```

You can find all of the information you will need by reading the *man* manpage, but here are some of the more useful switches:

-k str	search the manpages for instances of string str
-s #	search the appropriate manpage section
-a	shows all manpages that match, not just the first

2.6.2 Info

Besides *man*, there is also the *info* program. I personally don't use *info*, but it is very comprehensive and contains documentation for many programs, like emacs. However, *info* tends to be difficult to use and poorly organized. Regardless, much new UNIX documentation seems to be moving towards *info*. Included below is a table, adapted from UNIX Power Tools, of *info* commands (commands are case insensitive):

h	help using info
m	access a subtopic menu item
n	get to next related subtopic
p	get to the previous related subtopic
space	move forward
delete	move backward
C-l	redraw display
b	first page of document
?	list info commands
q	quit info
d	return to highest level of info topics
mEmacs*return*	To access the Emacs manual
s	search for string within current node

2.6.3 `/usr/share/doc/`

One last source of program documentation, at least in Linux, are what many programs will install to `/usr/share/doc`. Most programs will have a `README` and several other plain text files that are usually worth reading, along with a web-page. The web-page usually contains the program manual and is more detailed than a program's manpage. Generally, the only way to find out more about how a program works is to read its source code.

Chapter 3

Making UNIX Powerful

To effectively use UNIX there are a large number of commands that you need to know. The last chapter presented the very minimum number of commands that you would need to know in order to get by on the command line. While this may be all the more UNIX that you'll ever need, it is capable of so much more. This chapter is here to show you some of the more powerful and useful programs that you may want to use in your daily routines.

Rather than reproduce all of the information in each command's manpage this chapter will introduce the basic functionality of each command, like the previous chapter did. As always you can find out more information about each command by reading its respective manpage or searching on the web.

3.1 Logging In With `ssh`

Over the next four years *ssh* is going to become your new best friend. You have probably already used *ssh* to remotely and securely login to the CS Department computers, eliminating having to walk that arduous down hill journey to the Terminal Room to work on your homework. On the other hand, if you haven't yet discovered *ssh* prepare to be blown away!

ssh is a way of giving users a secure way to access and use a machine from just about anywhere: the moon, Sontag, or even Ohio. Assuming you have a viable network connection, and a user account, you should be able to login. *ssh* is secure in that it encrypts all of the traffic sent between you and the server, starting from when you first send your password, to when you log out.

If you run OS X, Linux, or BSD, you can invoke *ssh* from the command line (see section 1.2 if you are unfamiliar with how to get a shell) and login to a server like this:

```
% ssh USERNAME@URL.TO.SERVER
Password:
```

After answering the password riddle ("What is your password? What... is the air-speed velocity of an unladen swallow?") correctly you will then be presented with your lovely prompt. Note that nothing will appear when you type your password. *This is a security feature, and not a bug.*

As a more concrete example, say I wanted to remotely login to `knuth`. I would execute the following:

```
% ssh mkegel@knuth.cs.hmc.edu
Password:
```

Note that I had to specify the full URL to contact the server, in this case `knuth.cs.hmc.edu`. All HMC CS Department servers have the suffix `.cs.hmc.edu`.

Besides giving you access to a plain text terminal, *ssh* can also do X-Forwarding. That is, a program can be running on whichever computer you are logged into, but the program's GUI is displayed locally, which you can then interact with. To enable X-Forwarding use the `-Y` switch. Why `-Y` and not `-X`? Don't ask...better that some things are not known.

Note that X-Forwarding will only work if you are also running X11, and X-Forwarding is enabled on the server you are logging in to. Thus X-Forwarding will not work with OS X's *Terminal.app*. You first have to start *X11.app*, and then use *xterm* or an equivalent.

You can also use X-Forwarding on Windows, but you will need to download and install an X11 Server and have X-Forwarding enabled in PuTTY. For students who must work primarily from Windows, they should heavily consider installing and running an X11 Server.

One of the more advanced features of *ssh* is port forwarding. Through the magic of port forwarding you can bypass fire walls and other network obstacles. While beyond the scope of this manual, you should check it out.

> **What About Telnet?**
> You may be wondering whether you can login using telnet. The short answer is "for the love of god, NO!!". The longer answer is that you should never ever use telnet for anything other than a way to waste CPU cycles. This is because telnet does nothing to hide your password or encrypt any traffic that is sent between the two hosts. Since telnet has a complete lack of security, the CS department has decided to reject all telnet connections.

3.2 Moving Files Around

When you log in to any CS department machine you will always have access to the files in your home directory. However, there are times when you have to move files to or from your home directory. Discussed below are two different commands that can be used to transfer files between most hosts.

3.2.1 scp

scp stands for secure copy and allows you to transfer files between two hosts on a network. The secure part results from *scp* riding on top of *ssh* and because of this you will be forced to authenticate when you use *scp*. This also means that your files are encrypted as they travel across the network, which if you have sensitive data is a good thing indeed.

The basic syntax to copy a file from one host to another looks like this:

```
% scp user@host1:file1 user@host2:file2
```

Mercifully, *scp* allows you to omit the `user@host` stuff if you are copying to or from your local host. As an example, say you needed to copy your homework, `hw.cpp`, from your local machine to your home directory on knuth. Using *scp* you would type

```
% scp hw.cpp username@knuth.cs.hmc.edu:~
```

If you needed to copy an entire directory tree you would use the `-r` (recursive descent) switch.

3.2.2 sftp

sftp also rides on *ssh* like *scp*. Unlike *scp*, however, *sftp* is an interactive file transfer utility. If you have used an ftp client before then *sftp* should seem very familiar.

The following example shows how to connect to knuth as an sftp server:

```
% sftp username@knuth.cs.hmc.edu
(...Interactive Session Starts...)
```

The *sftp* utility is rather basic and doesn't provide any of the nifty features you may have grown used to while using the shell. Luckily there are some better sftp clients: `lftp` and `ncftp`. These two both provide a shell-like environment and are much easier to use than *sftp*.

3.3 Working With Files

3.3.1 Creating Files With `touch`

touch is a very useful program. It can be used either to create zero length files or to update the access time stamp of existing files. Filenames are supplied as arguments to *touch*. By default filenames supplied that do not exist will be created with zero length, and those that do will have their access time stamp updated. The following example demonstrates how to create the file `foo`

```
% touch foo
```

3.3.2 Finding Files With `locate`

If you've ever wondered where it was that you put that file, then *locate* can probably help you find it. Instead of taking the naive approach and searching the entire UNIX directory tree every time you want to locate something, *locate* searches a database containing all the filenames in the the directory tree. This makes access times much quicker, but also means that the database needs to be regularly updated. We have ours set to automatically update each day, so files created within the last day will not be in the database.

locate takes a file glob to search against. See section 2.5.1 for more information on file globs. If just a string is provided, like `"foo"`, *locate* will interpret this as being the file glob `"foo*"`. Otherwise *locate* will treat the file glob normally. Note that the file glob must either be escaped or surrounding in quotes to prevent the shell from trying to interpret the file glob.

When matching, *lcoate* matches your file glob against the *entire* name of every file in its database. Therefore, if you are searching for just one file, but have a directory with the same name, be prepared for lots of output.

The following examples demonstrate using *locate* to find a variety of files:

```
% locate "*Homework*06.pdf"
/home/user/documents/hw/Physics Homework 030506.pdf
% locate "Physics"
/home/user/documents/hw/Physics Homework 030506.pdf
/home/user/documents/hw/Physics Text.pdf
/home/user/Physics/lab files.tgz
...
```

3.3.3 Combining Files With `cat`

By design *cat* con*cat*enates smaller files together and prints the file contents to the terminal in the order the files were given. The following example shows *cat* printing out the contents of `bud`, `weis`, and `er`.

```
% cat bud weis er
A line from the file bud
A line from the file weis
A line from the file er
```

Note that *cat* doesn't actually create any new files, it only prints the ones given. When *cat* is run without any arguments it will read from stdin, and then reprint back to stdout.

cat is one of those programs you'll find yourself always using, and all it does is print the contents of files to stdout! See section 3.6.3 and section 3.6.4 to understand why this is important and useful.

3.3.4 Managing Space With `du`

The *du* program calculates and displays how much space a file or directory is using on disk. The default output, however, isn't very readable since *du* will display the number of 512 byte blocks that a file or directory is using. This is one of those archaic UNIX things and appears in other commands as well, so be aware!

The `-h` switch enables human readable output, causing *du* to display sizes in terms of Kilobytes, Megabytes, etc.

If you hand *du* a directory, it will print out sizes for each of its subdirectories as well. When you would rather know the total size of the directory the `-s` flag is quite useful—*du* will print a single entry for each file or directory supplied.

Here are some examples of *du* in action:

```
% du unixbook.pdf
123124   unixbook.pdf
% du -h unixbook.pdf
60M      unixbook.pdf
% du -sh courses/cs60
13M      courses/cs60
```

3.3.5 Archiving Files With `tar` and `gzip`

When working in UNIX you will no doubt need to archive files at some point. In Windows you have probably used a program like *WinZip* or *WinRar* to compress collections of files. In UNIX you will use a combination of programs: *tar* and *gzip*.

tar stands for Tape Archiver, so one can only guess how old this program is. When given some files or a directory (and the proper arguments) it will archive all the files and produce a single large file. It doesn't do any compression, however, which is where *gzip* comes in. *gzip* takes a single file and applies Lempel-Ziv compression, considerably reducing the file size. Because *tar* is so often used in conjunction with *gzip*, the GNU version of *tar* includes some switches for invoking *gzip* compression. Other versions of *tar* may not have these switches.

Here is an example of using *tar* to create a compressed archive of some directory `foo`:

```
% tar -czf my_archive.tgz *
```

Here is an example showing how to decompress an archive:

```
% tar -xvf another_archive.tar.gz
```

Note that `.tgz` and `.tar.gz` extensions to files are just convention and both mean the same thing. Try to use one or the other when creating your own archives. Also, you may also hear UNIX geeks refer to "tarballs". These too are just compressed *tar* archives.

`-c`	create archive
`-z`	use gzip to compress the archive
`-f`	read/write file; creates file my_archive.tgz in example
`-x`	extract files from archive; tar should detect compressed archives
`-v`	verbose output

3.3.6 Creating Links With `ln`

Under Windows you probably added shortcuts to commonly used applications to your desktop. These files weren't the applications themselves, they just pointed to the applications that you wanted to run. Links perform much the same function under UNIX.

There are actually two different types of links: "hard" links and "soft" links, or symlinks. They work much the same way, redirecting you to a different file, but have vastly different capabilities.

Symlinks are files that point the OS toward a different file or directory. When an application reads a symlink, the OS knows this and gives the application the correct data from the file being pointed to. To create a symlink you use *ln* with the `-s` switch. The following example would create a symlink called `link` that would point to `source`:

```
% ln -s source link
% ls -l link
lrwxr-xr-x   1 mkegel  mkegel  11 Jan 10 18:53 link -> source
```

Symlinks are very versatile; they can point to a directory and across filesystems. You will most often find yourself using symlinks, if you use links at all.

Hard links are much more simple than symlinks. Hard links create a new directory entry that then points to some file. This means that hard links do not exist as a separate file, like symlinks. This also makes hard links are much more limited than symlinks. You can only hard link to a file on the same filesystem that the link is being created on. By default *ln* creates hard links.

This example creates a hard link from `my_app2.1` to `my_app`:

```
% ln my_app2.1 my_app
```

The above example illustrates most succinctly what it is that hard links are used for. Hard links allow system administrators to keep around several versions of a program, while users use just the most recent version.

3.4 File Manipulation

You will spend the majority of your time creating, editing, deleting, and otherwise manipulating files in UNIX. So far I have partially covered all four tasks, but have mainly left file manipulation till now since there are so many ways different to manipulate files.

3.4.1 Sectioning Files With `head`, `tail`, and `split`

The simplest way to manipulate a file is to break it into sections. The three commands, *head*, *tail*, and *split* allow you to section a file in three different ways.

split takes a file and splits it up into chunks some number of lines long (1000 by default). The example below breaks up a long file into chunks 50 lines long (denoted by the `-l` switch):

```
% split -l 50 a_really_long_file
% ls
xaa    xab   xac
xad    xae   xaf
```

As you can see from the output of *ls*, *split* broke up `a_really_long_file` into six smaller files which are named in order. Of course, you should note that `xaf` may contain less than 50 lines.

To have *split* use a different prefix you can specify one after the filename, like this:

```
% split -l 50 a_really_long_file file
% ls
fileaa    fileab   fileac
filead    fileae   fileaf
```

Rather than dividing a file into chunks sometimes you just want the beginning or end of a file. The *head* and *tail* give you the beginning and end of a file (by default 10 lines), respectively. To get a different number of lines you would use the `-n` switch. Here is an example of using *tail* to view the latest contents of a log file:

```
% tail -n 5 /var/log/sys.log
(Imagine very nice output here)
(I'll get on a linux system at some point)
```

tail does much more than print the last few lines of a file. You can keep a file open and see data that is appended to a file with the `-f` switch. *tail* can also print *all but the last n lines*. To do this you use the `+X` flag, where `X` is some number. Both options are illustrated below:

```
% tail +5 /var/log/sys.log
(Lots of even more interesting output)
% tail -f /var/log/sys.log
(And more input until you hit C-c)
```

3.4.2 Counting With `wc`

The *wc* utility prints to `stdout` a count of the number of words, lines, characters, and bytes present in a file. The default output of *wc* includes a count of the lines, words, and bytes in a file printed out in that order. The following example demonstrates this behavior:

```
% wc some_file
     347    1861   12146
% cat some_file | wc
     347    1861   12146
```

As you can see the file `some_file` has 347 lines, 1861 words, and 12146 bytes. You can supply any number of file names to *wc*, or if there are none *wc* will read from `stdin` instead.

Here is a listing of some of the more common command line options:

Synopsis:
`wc [-clmw] [file ...]`
Switches:
`-c` Count number of bytes and print to `stdout`
`-l` Count number of lines and print to `stdout`
`-m` Count number of characters and print to `stdout`
`-w` Count number of words and print to `stdout`

3.4.3 Replacing Characters With `tr`

The *tr* utility allows you to translate one string of characters to another. It is more or less the equivalent of doing a global search and replace on a stream of text.

tr takes it's input from `stdin` and prints the text with the appropriate changes back to `stdout`. The basic syntax when using *tr* looks something like this:

```
% cat some_file | tr "STR_TO_MATCH" "REPLACE_WITH" > translated_file
```

This would replace all occurrences of `STR_TO_MATRCH` with the string `REPLACE_WITH` in the file `some_file`. When using *tr* to replace, you need to be aware that it will match strings inside other words. For example, if you wanted to replace `laughter` with `cackling` in a document that contained the word `manslaughter` you would end up with the word `manscackling` by mistake. This probably isn't what you meant to do.

Besides translation, you can also use *tr* to delete a string. For example, say you wanted to remove all the newlines in your CS70 homework before submitting it just to make the grader's appreciate nice formatting. The `-d` switch causes *tr* to remove the string given. The following snippet performs this function:

```
% cat homework.cc | tr -d "\n" > nonewlines.cc
```

I would not recommend that you do this more than once, unless you can get away with it.

3.4.4 Processing Text With `sort`, `cut`, `uniq`, `diff`, and `comm`

■ **sort**

For the times when you need to sort text there is the *sort* command. *sort* has all sort of functionality, which we'll only briefly cover here. More information about *sort* can be founding in Chapter 22 (Sorting) of UNIX Power Tools.

The default chunk of text that *sort* operates on is a single line. Within a line *sort* looks for fields. You can think of a field as a column of text. Technically, fields are groups of characters separated by whitespace, that is, some number of spaces or tab characters. *sort* begins counting fields at 0, and proceeds from left to right. For example, consider this example of a homework schedule:

```
% cat homework
stems           hw4         W   8:00am
e&m             pg.45       W   9:15am
cs70            hw5         M   12:00am
e&m_lab         lab         F   1:15pm
stems           hw5         M   8:00am
e&m             pg.62       M   9:15am
econ            reading     R   4:15pm
cs70            hw6         M   12:00am
discrete        hw5         W   2:45pm
```

The courses names in the above example are field 0, whereas the times would be field 3. If we were to run *sort* on the above example we would get this output:

```
% sort homework
cs70            hw5         M   12:00am
cs70            hw6         M   12:00am
discrete        hw5         W   2:45pm
e&m             pg.45       W   9:15am
e&m             pg.62       M   9:15am
e&m_lab         lab         F   1:15pm
econ            reading     R   4:15pm
stems           hw4         W   8:00am
stems           hw5         M   8:00am
```

Notice how `homework` was recursively sorted by field in alphabetical order. While this is generally the behavior you want, there will be times when you need to specify yourself what fields to sort by.

You specify a field with the following syntax: `+/-n` where `+n` means to starting sorting on field `n` and `-n` means to stop sorting on field `n`. To illustrate this, say we wanted to sort the file `homework` above by time first:

```
% sort +3 +0 -2 homework
cs70            hw5         M   12:00am
```

```
cs70            hw6         M   12:00am
e&m_lab         lab         F   1:15pm
discrete        hw5         W   2:45pm
econ            reading     R   4:15pm
stems           hw4         W   8:00am
stems           hw5         M   8:00am
e&m             pg.45       W   9:15am
e&m             pg.62       M   9:15am
```

Note: there is a `+1` implied after the `+0`. This is to keep *sort* sorting recursively; you need to explicitly specify a stop field if you do not want this behavior.

■ cut

Now that you know how to sort by field, let's learn how to rid yourself of the unneeded ones. The *cut* command allows you to cut by either column or field. In *cut* terminology a field is tab separated (unless specified otherwise with the `-d` switch), but does not mean quite the same thing as in *sort*. In *cut* the delimiters between fields are not greedy, so if one had two tabs in a row, the second tab would be considered a new field instead of being a delimiter. A column specifies the n^{th} column of text. Neither is probably quite what you meant. *cut* begins numbering fields and columns at 1.

Section 3.5.4 has more information about *awk*. If you want a better way to print fields, read up on *awk*.

You can select a single field/column or a range of fields/columns; commas separate values and hyphens define a range: e.g. 1,3-5,8-. Fields/columns are always printed in increasing order, so specifying 24,20 would print the same as 20,24.

To specify a field you use the `-d` switch; columns are specified with `-c`. The following example would print the 1st and 4th columns in the file `homework` above (assuming that `homework` has tab separated columns).

```
% cut -f1,4 homework
cs70                            12:00am
cs70                            12:00am
discrete                        2:45pm
e&m                             9:15am
e&m                             9:15am
e&m_lab                         1:15pm
econ                            4:15pm
stems                           8:00am
stems                           8:00am
```

Similarly we could grab the first 8 columns:

```
% cut -c1-8 homework
```

```
cs70
cs70
discrete
e&m
e&m
e&m_lab
econ
stems
stems
```

cut's behavior becomes much tolerable as you learn about Regular Expressions in section 3.4.5.

Below is a good example of using *cut*:

```
% cut -d: -f1,5 /etc/passwd
(...)
```

Can you guess what will be printed?[1]

■ uniq

uniq detects and removes duplicate adjacent lines in a file or input stream. For example, say we have the following file:

```
% cat somefile
one
one
two
three
one
```

Running *uniq* we would get the following output:

```
% uniq somefile
one
two
three
one
```

By sorting the file first, all duplicates can be removed. Some useful switches to *uniq* are: `-c` which precedes the output line with a count of the number of duplicates removed, and `-f num` which begins comparing on the `num`th field (`sort` semantics; counting starts at 1).

■ diff

[1] Answer: the user names and real names of users will be printed.

diff is one of the basic UNIX utilities. Its main purpose is to compare two files line by line and display. For example, say we've been using version control on our large and complicated networks project. Today, you see that a bug was introduced that you had squashed last week. So what changed between last week and today that introduced the bug. Because *diff* shows you the differences between the two files, you can easily spot the error. *diff* works on any sort of plain text file, but is not meant for common binary data. For non-plain text data a utility called *cmp* is what you're looking for.

Here is an example of *diff*ing two files:

```
% cat a
one
five
three
% cat b
one
four
% diff a b
2,3c2,3
< five
< three
---
> four
> two
```

As you can see, reading *diff* output can be a bit confusing. Here are the basics: lines with a "<" are from file `a`; lines with a ">" are from file `b`; and the change marker (`2,3c2,3`) and the `---` marker make up a section that indicates exactly what and where the changes are between the two files.

More information about *diff* can be found in *UNIX Power Tools*, in the *info* database, and on the internet.

■ comm

comm is similar to *diff*, but instead of displaying what changes exist between files in an obscure format, it graphically shows you which lines are different. *comm* produces three columns of output: lines only in file 1; lines only in file 2; and lines in both files. Here is the output of running *comm* on the files `a` and `b` as defined above:

```
% comm a b
                one
five
        four
three
```

There is one switch of the form `-n` where `n` is the column number (1, 2, or 3). This switch suppresses output of the indicated column. You can suppress multiple columns.

3.4.5 Processing Text With Regular Expressions

Up until now you've just been playing with small arms, but now it's time to break out the heavy artillery. In terms of raw power, *grep* and *sed* are the two most powerful utilities (besides full blown languages, of course) that you can use on the command-line. And knowing how to take full control of both programs will make you the geek's geek. Okay, that's enough hype. But rather than get into the technical details, you should probably be aware of what each program does.

grep allows you to apply a pattern to an input stream, and print the lines of text that match the pattern given. While this might sound boring, being able to pull out lines of text is actually extremely important. For example, say you've marked buggy sections in your code with a "FIXME" statement. If you're working on a large project, then chances are that you'll forget to go back in fix all the FIXMEs. *grep* allows you to see what files, on what lines, in what context, contain these, or other, statements.

sed is very similar to *grep* in that it takes an input stream and a pattern. The output that *sed* produces, however, is closer to what *tr* does. *tr* does a simple translation of one string to another. In the same vein, *sed* can also replace strings in the stream, but based rather on the contents of the specified pattern instead of a simple string to which *tr* is limited. As a motivating example, say that you wanted to parse a phone number for correctness: there should be nine digits, organized into two groups of three and then a group of four, and punctuation should be ignored. *sed* would allow you to easily perform this task, and subsequently format the output in a sane way.

So far I've used the term pattern, without really defining what it is. You've more than likely already had a chance to use (or creatively abuse) the file globs that were discussed in section 2.5.1. To refresh your memory, a file glob allows you to specify files by their characteristics, that is, what string they start with, whether they have a digit as the third letter, etc. In Computer Science (finally real CS!) a file glob would be a kind of "regular expression", though Unix users tend to reserve that term for Perl or Posix patterns.

So what does this have to do with *sed* and *grep* and patterns? Well, 'pattern' is really just another name for 'regular expression', except in this case you aren't writing file globs but rather UNIX regex's (regex is short for regular expression). This is what makes *grep* and *sed* so powerful, since as you'll learn in CS60, regex's (aka. DFAs) are one of the fundamental computing tools.

Unfortunately, explaining how regex's work, and how to write them well, is not easy. Fortunately there are some very good treatments of regular expressions. The first exists in the book *Programming Perl*. Perl regex's define the standard by which all other regular expressions are evaluated, and this book explains in a thorough and clear manner how to write Perl regex's. Fear not, since because Perl

regex's have become a de-facto standard they can be written most anywhere, in any language, and be expected to work. In fact, once you learn regular expressions in one setting that knowledge can be easily transferred. The other authoritative and comprehensible source is the book *Mastering Regular Expressions*. This book delves more into the theory behind regex's, but is still well worth your time. But since dead trees and charcoal are cheap, here is my explanation of how to write simple regular expressions [2].

You will usually define a regex between slashes: /regex/. *grep* and *sed*, however, each use a slightly different syntax, but we'll discuss that when we get there. The simplest regex to write is just a simple string. As you would expect, the only thing that matches a string is the string itself. While not flashy, it is what allows you to find "FIXME" in source code comments.

The example below uses *grep* to locate lines containing the string FIXME:

```
% grep "FIXME" *.cc
```

To get the same functionality using *sed* requires some magic, which we'll get to later.

Back to regex's, you often need to specify and match against more than a single string. The OR operator is the pipe (`|`), and it allows you to specify multiple expressions, in this case strings, to match against. The regex `/foo|bar/` would match either the string "foo" or the string "bar".

The `^` operator is kind of odd—it doesn't represent any character. Rather, it represents the nonexistence of a character: a sort of imaginary thing that exists between the beginning of a line and the first character on that line. So, `/^foo/` will match lines that start with 'foo', but not one that starts with 'foo' or 'bar' or 'Foo'.

The $ operator is like `^`, but for the end of a line. `/foo$/` matches strings that end with 'foo'.

Parenthese provide grouping: `/foo(bar|baz)/` will match either foobar or foobaz. They also provide the ability to later extract what was contained in the parentheses, which is called "capturing".

Brackets match character classes: `[abcd]` will match one character from that group. You can also do ranges: `[a-zA-Z0-9]` does what you think it should. One important note: `^` inside a character class has some tricks. If it's the first thing in a character class, it implies 'match as if the character class was everything EXCEPT what's typed here'. So, `[^ ]` would match any non-space character. If it's placed anywhere but in the first position in the character class, `^` behaves just as any other entry in the character class does.

There are various handy prebuilt classes in PCRE and other good regex libraries. POSIX specifies things like `[:alnum:]` for alphanumerics, `[:digit:]` for 0-9, etc, but those haven't gained widespread use, since no one cares about POSIX. Perl-style classes are much more common in scripting languages. Perl uses `\s`

[2] OK, so it isn't exactly mine. The rest of this section has been adapted from a handout on regular expressions by Marshall Pierce.

for whitespace, `\S` for non-whitespace, `\w` for for 'word' characters (alphanumerics and '_'), `\W` for non-word characters, `\d` for digits, etc. There are tons of these things, but I tend to use only `\w`, `\W`, `\s` and `\S`. See page 161 of *Programming Perl* (aka "The Camel") for all of them. There are also Unicode properties. `\p{InIdeographicDescriptionCharacters}` sounds unpleasant, so let's just skip that. See p. 170 in the Camel if you enjoy pain. I mean, if you enjoy Unicode properties.

While I'm on the topic of odd metacharacters, `^`, `$` and `.` (see below) suffer from various complications when you're dealing with multi-line patterns. Investigate `\a`, `\A`, `\z`, and `\Z` on p. 179 of the Camel if you need to work with that, as well as the 's' and 'm' flags on p. 147. This stuff comes up less often that you'd think. I've never had to deal with it myself.

A dot (`.`) stands for any character at all (except a newline (usually—see above)). It also has a special meaning in character classes, or rather, it has absolutely no special meaning in a character class: it's just a plain old period. (Using 'any character at all' in a character class doesn't make much sense.)

The star operator (`*`) means to match 0 or more repetitions of the previous character. `a*` matches '', 'a', 'aaaaaa', etc.

The plus operator (`+`) is like `*`, except it matches 1 or more repetitions: `a+` matches 'a', 'aaaaa'...

`{min, max}` let you specify numerically the minimum and maximum number of allowed matches. `{2,}` will match 2 or more repetitions. `{2,8}` matches from two to eight repetitions.

A question mark (?) means 'one or zero'. `a?` matches '' or 'a'.

`*`, `+`, `{}` and `?` can all be used on groupings and character classes, as well: `[aeiou]+` is any non-zero string of vowels, and `(foo|bar)?` matches `' '`, 'foo', and 'bar'.

It's important to note that `*`, `+` and `{}` are 'greedy'. As an example, suppose we have a string 'aaabaaaa'. `a*` would technically match against nothing at all, since a zero-length string technically is allowed. However, its default behavior is to match as much as possible, so it'll grab 'aaa'. Why didn't it grab the second string of a's? regexes are greedy, but not psychic... it'll return the first greedy match that satisfies its conditions. You can change the greed of `*`, `+` and `{}` with the `?` suffix. `/a+?/` will match just one 'a' on our pattern, `/a*?/` would match `' '`, and `/a{3,4}?/` would match 'aaa'.

One thing you're probably wondering by now is what to do if the string you want to match contains a period or a plus sign. You can strip these characters of their special meaning by escaping them with a backslash. `/\(\/\.\)/` will match '(\)'.

Another handy trick is that in some situations (basically, if you're using Perl), you can use something else instead of '/' as your regex delimiter. If you're doing something that involves a lot of paths, such as `'/usr/local/bin'`, you don't want to have to escape that into `'\/usr\/local\/bin'`, etc, so you can just choose to use a different delimiter, like `!` or `#`, to produces regexes like `#/usr/local/bin#`. I don't tend to use this much, but you might see it in other people's scripts.

You can also pass flags to regular expressions to control their behavior. `/a/i` will match both a and A, due to the case Insensitive flag. `i` and `g` are the only flags I ever use. `g` means 'global', and is only applicable to replacements, which I'll cover in a bit. You can 'cloister' arguments such as the `i` flag to a certain chunk of your regex with this syntax: `(?_:PATTERN)`, replacing the `_` with the flag (or flags) that you want. You can subtract modifiers, too: this will apply `i`, but cancel `x` (which is a flag that lets you be a little more carefree with your whitespace inside patterns), to the subpattern `(foo|asdf)`: `(?i-x:(foo|asdf))`.

There's only one major matching tool left to cover - lookaround assertions. These make the match succeed if the thing you're looking for does (or doesn't) exist in front (or behind) of where you make the assertion. They come in the four obvious: positive lookahead `(?=PATTERN)`, positive lookbehind `(?<=PATTERN)`, negative lookahead `(?!PATTERN)`, and negative lookbehind `(?<!PATTERN)`. Here's an example that will succeed if 'bar' exists somewhere in the string after 'foo': `/foo(?=bar)/`.

At this point, you've seen almost everything (well, everything that matters) about how to see if a string matches a pattern. Now comes the easy part: extracting out the parts of the string that you care about. This is where the parentheses' capturing comes into play. Suppose we have str = 'foo bar'. The pattern `/(\w+) (\w+)/` will clearly match str. The handy part is that Perl, Ruby, and other languages will set variables for you so that you can access the parts grouped in `()`. In Perl's and Ruby's cases, the variable `$1` will contain 'foo' and the variable `$2` will contain 'bar'. If you had `((\w+) (\w+))`, `$1` would be 'foo bar', `$2` 'foo', etc. (Note that if you're using stuff you've previously captured inside the regular expression, you refer to those captured strings as `\1`, `\2`, etc.) Even if you don't capture anything, you still get a few convenience variables: `$&` is the matched part, `$‘` is the part before it, and `$’` is the part after it.

The really nifty part is that you can change the string around by substituting in things based on your captured matches. This is usually denoted `s/PATTERN/REPLACEMENT/`. Here's an example that draws on much of what we've seen before. Let's say you have a set of strings of the pattern ¡var¿ = ¡value1 value2 value3 etc¿, but you want to have the value list be comma-delimited instead of space-delimited. Here's a first attempt:

```
line =~ /^(\w+\s*=\s*)(.+)$/
valuelist = $2
valuelist =~ s/(\w+)\s+(?=\w+))/$1, /
```

However, this will only produce "value1, value2 value3 etc". Remember the 'g' flag? That makes substitution work Globally—that is, instead of stopping after the first successful replacement, it'll keep trying. So, the one we want is `s/(\w+)\s+(?=\w+))/$1, /g`. (That's the Perl version. For the curious, the Ruby syntax for that would be `.gsub(/(\w+)\s+(?=\w+)/, '\1,')`, and the syntax for the earlier version would be `.sub(/(\w+)\s+(?=\w+)/, '\1, ')`)

If you want to use (simple) regular expressions to filter out stuff, *grep* is very handy. Here, I'm using it to filter out lines that match 'foo' from the file 'file1'. *grep*

is really limited, though—don't expect much more than the basics like `^`, `[]`, or `*`. See grep(1) for more details.

```
% grep foo file1
```

I tend to use grep most as a filter, though.

```
% command_that_spits_a_lot_of_output | grep 'string I want to look for'
```

sed is another command-line tool that uses regular expressions. *sed* is generally used for substitutions. There are two handy commands for *sed*: `s`(ubstitute) and `d`(elete).

By default, *sed* applies the commands to every line. You can change this by specifying an address pattern to use. Patterns can be line numbers, regexes, or things like `$`.

No address implies that *sed* should apply commands to every line. One address implies action only on lines that match the address string; two addresses imply action only on lines between the first line that matches the first address and the first line that matches the second address. You can negate addresses by appending a `!`.

So, just 'd', without an address, will delete every line. `1d` will delete just the first line. `1,5d` will delete lines 1-5. `$d` will delete the last line. This is not the same as `$ =` end of line. `/^$/d` will delete every blank line. `1,/^$/d` will delete from the first line until the first blank line.

All the addressing stuff also applies to the 's' command, but to clarify exactly what 's' does, we won't mix addressing and s.

`s/tree/cat/` replaces the first instance of 'tree' with 'cat'. `s/tree/cat/g` replaces every instance of tree with cat (g for global). `s/tree/(&)/g` replaces every instance of tree with (tree). The ampersand (`&`) references the entire match. `s/tr\(ee\)/\1/` replaces 'tree' with 'ee'. `s/\(tr\)\(ee\)/\2\1/` replaces 'tree' with 'eetr'. `s/\(tr\)\(ee\)/\2\2\2\1/` replaces 'tree' with 'eeeeeetr'. `s/<\([^>]\)>\(.*\)<\/\1>/[\1]\2\[\/\1]/` replaces `'<b>foo</b>'` with `'[b]foo[/b]'`

Note that unlike Perl & Co., *sed* wants you to escape parentheses to make them capture, rather than the other way around.

3.4.6 Searching Through Files With `grep`

3.4.7 Mangling Text With `sed`

3.4.8 Mass Manipulation With `find`

If you know how to use *find*, you may well never need to learn the complexities of shell scripting or Perl/Python/Ruby. The *find* program is extremely versatile, but is best suited to processing large numbers of files. For example, you may need to convert all the `jpegs` in a directory hierarchy to `bmps`. If you have more than a dozen or so images, it quickly becomes an impossible task unless it is automated.

Let's start with a basic example, such as using *find* to just print a list of files in a directory:

```
% find . -print
./hmclogo.eps
./hmclogo.pdf
./hmcseal.eps
./hmcseal.pdf
./Preface.aux
./Preface.log
./Preface.tex
```

find can take a number of arguments, however, the first argument to *find* is usually the directory that *find* should start at, in this case `.`, or dot. After that you provide *find* with an expression to evaluate in the form of arguments. This expression is evaluated on *every* file starting at the directory you gave and recursing down. The `-print` expression always evaluates to true, and causes *find* to print the filename.

Now for something more complex—say we only want those files that begin with "hmc". For this we would use the `-name` expression, like so:

```
% find . -name "hmc*" -print
./hmclogo.eps
./hmclogo.pdf
./hmcseal.eps
./hmcseal.pdf
```

Notice that the `-name` expression took a file glob, so we could have declared something much more complex. See the *find* manpage for which globs are supported. The `-name` expression only matches the pattern against the filename, to match against the file's path you would instead use `-path`.

find offers the ability to logically combine expressions. The simplest is the logical AND operator. In the above example we made implicit use of this operator by juxtaposing the `-name` and `-print` expressions. When both evaluated true, the designated action (here printing) would occur. You can also be more explicit. For example we could have written:

```
% find . -name "hmc*" -and -print
./hmclogo.eps
./hmclogo.pdf
./hmcseal.eps
./hmcseal.pdf
```

You can use `-or` as the logical OR operator. Both the AND and OR operators short circuit to avoid unnecessary processing.

Expressions can also be grouped. To group expressions you surround them in parentheses, which usually need to be shell-escaped. With grouping the example would have been:

```
% find . \( -name "hmc*" -and -print \)
./hmclogo.eps
./hmclogo.pdf
./hmcseal.eps
./hmcseal.pdf
```

The `-not` expression provides the logical NOT operator, so we could have matched every file not meeting the above expression like this:

```
% find . -not -name "hmc*" -print
./Preface.aux
./Preface.log
./Preface.tex
```

The two most powerful expressions *find* offers are `-exec` and `-execdir`. These each allow you to declare some command that will be executed on each file matched, with the difference being that `-execdir` will execute from the directory holding the file. This behavior of `-execdir`s is useful for when you need to manipulate a file and the file's path matters, live when you move a file. Otherwise everything else about the two expressions are the same, including how to use them.

As an example of using these expressions, imagine that we need to remove all the files named `lock`. This operation is complicated by the fact that these files are themselves located inside specially named directories: `.svn`. For those whom this might be familiar, this is just a motivating example. Here's what we might try:

```
% find . -name ".svn" -exec rm {}/lock \;
```

First, we use the `-name` expression to match on the correct directory name. Second, we use the `-exec` expression to execute *rm*. When executing a utility, you often need to refer to the file itself—the "{}" string is expanded by *find* to the filename. The `-exec` expression must be closed by a semi-colon, as you can see at the end of the line. You can also include arguments to the executed utility, and manipulate the "{}" token as shown above.

Synopsis:

```
find [path ...]  expression
```

Switches:

-name GLOB	evaluates true when filename matches GLOB
-iname GLOB	case insensitive version of `-name`
-path GLOB	evaluates GLOB on file's path; true when matches
-ipath GLOB	case insensitive version of `-path`
-print	print filename; always true
-exec CMD [ARGS]	evaluate CMD on file; true when CMD returns 0
-execdir CMD [ARGS]	performs `-exec` from file's directory
-delete	deletes the file; always true
-group GROUP	true when file is in group GROUP
-perm MODE	checks file permissions against MODE
-size N	checks file size; N is number of 512-byte blocks
-user USER	true when file is owned by USER

3.5 Arbitrary File Manipulation

The programs introduced so far will be able to handle 90% of the things you will want to do. But there are times when these utilities fail and you need something completely custom. This is the point at which you seriously start to think about writing a program to solve your problem. But what language do you choose?

The C, C++, and Java languages are all well worth knowing. Most code in industry is being written in the latter two, though Java seems to be falling in popularity. Most UNIX code has been written in C, so it will be around essentially forever. There are times, though, when you want to sit down and code something simple without dealing with the hassle of coding in C/C++/Java.

Perl, Python, and Ruby each offer a simple, easy to learn syntax, a generous and powerful standard library, many additional libraries, and memory management. I would highly recommend that you learn at least one of the discussed languages. It should take you a day or two to learn the syntax and an adequate amount of the standard library, but the time saved in future projects will be considerable. You may even have fun programming in these languages!

The most important feature of Perl, Python, and Ruby is that they are executed by an interpreter, and thus are platform-independent. Interpreters are separate programs compiled for the specific operating system and architecture you are running on, and are responsible for executing the code you have written. In most cases, the only code that has to be written for a specific OS is the interpreter. Your code can be written to run just about anywhere with no changes. This is similar to the way that Java works; Python even compiles down to byte-code. You can find versions of Python, Perl, and Ruby that run under Linux, Windows, OS X, and many others.

While being interpreted makes these languages platform independent, you do suffer a slight performance penalty. If performance is a key factor to your project (i.e. you are doing graphics, or scientific computing), then you should stick with a language compiled to machine code, like C/C++, otherwise use one of these languages. The time saved in using one of these languages will far outweigh the lesser performance.

3.5.1 Perl

Perl is the oldest, best known, and by far the most difficult to read interpreted language. Created by Larry Wall in 1987, Perl is now more or less standard in all UNIX installations.

Perl is excellent at processing text, mainly because of its excellent regular expression support. Perl regular expressions have become the de facto standard and are supported in both Python and Ruby. Beyond this application, though, you should probably consider a different language. Perl has only very rudimentary

support for objects and is easy to make completely unreadable unless you have a very strict coding style. Perl is best suited to small projects, on the order of one to two hundred lines of code. Anything larger than this can be impossible to maintain. Perl 6, which should be out sometime before Christ's second coming, promises to fix many of these issues, but may come too late to the game.

If you want to learn more about Perl the best resource is *Programming Perl* (a.k.a. The Camel book) published by O'Reilly. There are also several very good websites: `www.perl.org` and `www.perl.com`.

3.5.2 Python

Python is a newer language than Perl and corrects some of its follies. Python is a strongly typed and completely object-oriented language, but manages to be friendly in almost every way. The language is much easier to read than Perl and almost feels natural to write. Like Perl, it offers superb handling of strings and lists, enough to make me green with envy after programming in C.

Python does have its faults. The language depends on the spacing of statements to delimit blocks of code, rather than using typed delimiters (think brackets surrounding functions in C). While it does enforce a good indentation style, a funky text editor can completely corrupt Python code if you are not careful. Sharing Python code can also be a problem because of this, particularly if you mix tab characters and space characters.

The trait I find most annoying about Python is that class methods must have a self variable to refer to the object being passed into the method. Rather than hide this layer, Python has made it the job of the programmer. This is needless complexity.

Python seems to be good for large projects. You can easily break up Python code across many different source files. Finally, the number of development tools (IDEs, shells, editors, etc) make Python a fun language to develop in.

To find out more about Python checkout: `www.python.org`. The website has everything that you'll need to learn Python. The Python Tutorial is enough to teach anyone and the Global Module Index serves as an excellent reference later on. The book *Beginning Python: From Novice to Professional* has also garnered good reviews and may be a worthy addition to your library.

3.5.3 Ruby

Ruby is a Japanese import and is the newest of the three. Like Python Ruby is completely object-oriented, with everything being an object of one kind or another. The most interesting feature of Ruby is its excellent use of lambda forms. Otherwise Ruby is fairly standard, drawing a lot of inspiration from Perl without inheriting any of the nastiness. For many different reasons Ruby may be the best language of the three to learn.

To find our more about Ruby checkout these websites: `www.rubydoc.org` and `www.ruby-lang.org/en`. The book *Programming Ruby* is also an excellent resource.

3.5.4 Awk

The *awk* program has been included in this really only for reasons of completeness. If you know any languages in the scripting trifecta of Python, Perl, and Ruby then there is really no reason to learn *awk*. In fact, in general, there is really no reason to learn *awk*. But if you want one more language under your belt, or have some specific need, then you can always learn *awk*.

awk is an imperative scripting language, with no object-oriented features, and comes in three flavors: *gawk*, *nawk*, and *awk*. The first two are compatible with the last, but offer several new features. *gawk* is the open source GNU implementation.

awk is most useful, and easiest to use, when working with tabular data, that is, data organized into distinct rows and columns. One might say that this is the only real use *awk* has anymore. An an example, say you wanted the first column (columns are separated by spaces or tabs) of some file. The following line accomplishes that:

```
% awk '{print $1}' filename
```

Moving much beyond this is neither trivial nor useful. For more information about using and programming in *awk* see the CS Department QREF covering *awk*.

3.6 Working With Processes

Infinite loops. Bugs. Broken Parsers. Misspelled words. All of these things, and many more, show up in the code that you'll write. Of course, you won't actually know it until you run your code, and that's where this section comes in. You've already learned a bit about job control (section 2.5.4), but there is much more to process management than that. The following sections explain what you'll need to know to manage processes in ways you haven't even dreamed of.

3.6.1 Viewing Processes With `top` and `ps`

The *top* program is a command line utility that displays all of the information about each of the processes currently running on the system as well as system wide information: CPU usage, load averages, memory usage, etc. It is analogous to Activity Monitor on OSX and FIXME on Windows. By default *top* sorts processes by the amount of CPU usage each one is taking, with CPU hogs going at the top of the list. You can quit *top* by pressing the `q` key, or find help by pressing the `h` key.

For now the most important thing to pay attention to in *top* is the first column. This column shows the PID, or Process ID, of the process in that row. Every process on the system has a unique PID and when you need to refer to a process you'll most often need to know its PID.

The *ps* program is similar to top, but doesn't do any of the live updating that *top* does. Instead it prints a list of processes, and some information about each, to your terminal. *ps* run alone will just print out the processes that have been started

by that particular shell. To have *ps* show you all the processes on the system you'll need to use the `-a`, `-u`, and `-x` switches. *ps* will also print out a processes PID. *ps* is nice for when you need to be able to search through the sometimes considerable list of running processes and find the PID for a particular process.

3.6.2 Killing Processes With `kill`

When a process refuses to stop you have to kill it. Of course you can only kill processes that you own, so don't think that you'll be able to crash the system or wreak havoc with your friends web browser. You need root for that :-).

To kill a process you need its PID. See the previous section to find out how to get a process's PID if you are unsure. You then supply the PID as an argument to *kill*. For example if we needed to kill chunky string because it's stuck in an infinite loop we would find the PID by using *ps*, and then use *kill*:

```
% ps
  PID  TT  STAT      TIME COMMAND
 3027  p1  Rs     0:02.49 -zsh
 2575  p4  Ss+    0:03.48 -zsh
 23156 p4  Rs     7:23:12 chunkystring
% kill 23156
[1]  + terminated  chunkystring
```

When a process will not die with a simple *kill* command you may have to use the `-9` switch. This switch indicates that you "really mean it this time." Also checkout the *killall* command, as you may find it useful for when you need to kill multiple processes at once.

3.6.3 Redirection

Up till now you've just been reading the output of programs on your terminal. But what if you wanted to save the output of a program? How would you do that? By using the shell, silly.

Before continuing there are some technical details that need to be covered. Programs can actually have two different kinds of output both of which are printed to the terminal by default. One kind is just normal output. The other contains error or debug messages. It is output that you would not otherwise want. The first gets referred to as Standard Out, and is abbreviated `stdout`. The second is referred to as Standard Error, and is abbreviated `stderr`. There is only one input, Standard In, which is abbreviated `stdin`.

For a more concrete example, if you have dealt with C++ before then you know that you write to the terminal on `stdout` using `std::cout`. To write to `stderr` you would use `std::cerr` and to get input you would read from `std::cin`, or `stdin`. From the perspective of the program what you're actually reading from and writing to are files. Keep this in mind as we discuss *redirection*.

When you work with the `std*` of a program, you are said to be redirecting the input/output. The simplest way is to redirect output to a file. For example, we could redirect `stdout` of *echo* to a file. To do this we place a > after the command followed by a filename, like this:

```
% echo "This string get redirected" > some_text.txt
```

Notice that nothing gets printed to the terminal. Note: we have things set so that existing files won't get *clobbered*, that is written over when you redirect output, so you may need to write to a new file or erase the old one. You can change this by unsetting the `NOCLOBBER` environment variable.

You can also append `stdout` to a file, that is add things to the end of a file. The append operator is >>. If we had wanted to append the output of *echo*, the previous example could have been:

```
% echo "Appended text" >> some_text.txt
```

A program can read its `stdin` from a file using the < operator, like this:

```
% cat < some_file.txt
This text was in some_file.txt
```

Working with `stderr` is very similar, and each of the standard operators are outlined and explained in the table below:

`cmd < file`	redirect `stdin` to `file`
`cmd > file`	redirect `stdout` to `file`
`cmd >> file`	redirect and append `stdout` to `file`
`cmd >& file`	redirect `stderr` to `file`
`cmd >>& file`	redirect and append `stderr` to `file`

Getting Rid of Output

There are some special "devices" that the kernel provides to users that you may find yourself needing. The device `/dev/null` acts as a sink. It is a place where you can send output to oblivion. Reading from `/dev/null` will just return an EOF. The device `/dev/zero` provides an infinite stream of zeros and is useful for when you need some source of output. The devices `/dev/random` and `/dev/urandom` both provide random output, with `/dev/random` being *truly* random.

3.6.4 Pipelining

Programs in UNIX are written to do one thing and do it well. More powerful and complex programs can then be constructed LEGO-style from smaller programs. One way this is accomplished is through *pipelining* programs.

You learned about redirecting `stdout` to a file in the last section. More often, however, you want to use that output as the input to another program. The shell allows you to pipe output from one program to another using the pipe | character, like so:

```
% cmd1 | cmd2 | cmd3 | cmd4 | ... | cmdN
```

To pipe `stderr` you would use `|&`, like so:

```
% cmd1 |& cmd2
```

When programs are used like this on the command line, they are often referred to as *filters*. That is, they filter the output and format it in some fashion. Learning to use filters is key to becoming a UNIX Power User.

Here two concrete examples of using filters:

```
% cat some_file | grep -n "some_string"
7: line with some_string in it
24: another line with some_string
% who | wc -l
        15
```

The first example finds instances of `some_string` in a file using *grep*. The second example pipes the output of *who*, which prints who is currently on the system and when they logged in, to *wc* which counts the number of users (actually the number of lines) and prints the value.

good pipelining examples - word frequency

3.6.5 Backticks

When you quote a string with backticks, the shell interprets the quoted part as if it were a command and replaces the string with the output of a command. This example illustrates the concept:

```
% emacs ‘grep -l FIXME *.c‘
```

Here the shell run *grep*, which prints out those source files that have the string FIXME. The output of *grep* then replaces the backticks and everything quoted inside. Finally, *emacs* opens each of the files.

Backticks are not often used on the command line since more often programs will take input on `stdin`. Instead, they are more often used in shell scripting. Knowing this syntax is useful just in case.

3.6.6 Keeping Things Running With `screen`

Marshall should write this section.

3.7 Customizing Your Shell

There are many things you can do to customize your shell or control its behavior. Much of this behavior is controlled by the options you set in the file `.zshrc`, which is located in your home directory. When you login *zsh* reads this file and executes whatever commands have been placed there. Other shells have different startup files: *bash* reads `.profile` and *tcsh* reads `.cshrc`.

To get an accurate idea of what your `.zshrc` does you ought to read through it now. Open it up in *emacs* or with *less*. The file is heavily commented and the result of each line is fully explained. Some of the things set in your `.zshrc` include: your `umask`, the `PATH` variable, aliases, your prompt, turning tab completion ON, and various *zsh* options.

zsh contains far too many options and is far to complex for it all to be explained here. However, learning how to fully utilize your shell, and customize it to your exact needs and style, is very valuable. Towards that end here are some references that you may want to read:

- From Bash to Z Shell
- `www.zsh.org`

3.7.1 Environment Variables

Environment variables control much of the behavior of your shell and the programs that it runs. They control which directories are searched when you type a command (`PATH`); which editor programs will use (`EDITOR`); and even who you are (`UID`). Environment variables are maintained by the shell, and every program that you execute from the shell has access to these variables. There are also shell variables that only the shell has access to, but you needn't worry about them. By convention environment variables are named in all uppercase letters, whereas shell variables are named in all lowercase. If you create new environment variables be sure to follow convention.

The following example prints the value of the environment variable `EDITOR`:

```
% echo $EDITOR
emacs
```

When you precede a string with a dollar sign (`$`) the shell interprets the string as a variable and inserts the value of variable. If the string doesn't correspond to a variable the shell inserts nothing. Its a bit like using C pointers. The variable can be used as part of the arguments to another command, or you can use the *echo* program to print the value of a variable, like in the example above.

To create an environment variable you would use the *export* program. For example, if you wanted to change the value of `EDITOR` to something like *vi* you would type:

```
% export EDITOR="vi"
```

If you would like to permanently set an environment variable you would add the `export` line above to `~/.zshrc`. Spacing is important when declaring variables—make sure you follow the example above.

The CS Department has a bunch of variables that we set by default for accounts using the *localenv2* program. This program is executed by your shell when you log in. You can change these defaults just by adding entries for the appropriate variables to your `.zshrc`. If you would like to set all of your own values then remove the *localenv2* entry from your `.zshrc`. We would not recommend this, but you do have the option.

3.7.2 Changing Your Shell Prompt

The shell does more than just execute your commands, as I'm sure you've noticed. An ever present reminder of this is your shell prompt. It can be as big and gaudy as you'd like (Google for Phil's zsh prompt), or minimal and simplistic, like our stoic `%`. I prefer something in-between the two extremes. This is the kind of prompt you'll have when you login for the very first time. An example of the default knuth prompt is shown below:

```
21:14:55 [mkegel@shadow:~/Documents/ACM Unix Book ]
(5) %
```

It shows you the time, your current username, the server you are logged into, the current working directory, and the current command number. The default prompt is ok, but not everyone likes the way it's setup.

To change your shell prompt you modify the `PROMPT` and `RPROMPT` environment variables. *zsh* is fairly sophisticated and supports both regular left-aligned prompts (like the default, stored in `PROMPT`) and right-aligned prompts (stored in `RPROMPT`). Right-aligned prompts can be both useful and aesthetically pleasing.

To get things like the date and current working directory you use escape codes. You can also add color with the appropriate escape codes. Below are some of the more common escape codes. You will need to consult the *zsh* manual for a complete listing. The following is taken from the book From Bash to Z Shell:

%B and %b	start and stop bold mode
%S and %s	start and stop standout mode
%U and %u	start and stop underline mode
%~	Full name of current directory
%m	Host (machine) name
%n	Username
%T	Time in the 24-hour format, such as 18:20

Once you've decided what you'd like to have in your prompt, you set the appropriate environment variables. The following example changes the prompt to have the current user and time on the left side, and the current working directory on the right:

```
% export PROMPT="some set of escape codes"
[12:34] mkegel % export RPROMPT="some other set of codes"
[12:35] mkegel %                                                    [~]
```

To make the changes to your prompt permanent you would add both `export` lines to `~/.zshrc`.

3.7.3 Changing Your `PATH`

One important environment variable is `PATH`. This variable contains a list of directories the shell will search when you type a command. The shell will then execute the first command that matches what you've typed.

The directories that `PATH` holds are colon separated. Here is an example of what `PATH` could contain:

```
% echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin:/opt/local/bin:/usr/X11R6/bin
```

To add directories to `PATH` from the command line you will want to append to the current `PATH`. Prepending is allowed but is generally a Bad Idea ©. To add the directory `~/bin` to `PATH` you would execute:

```
% export PATH=$PATH:~/bin
% echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin:/opt/local/bin:/usr/X11R6/bin:~/bin
```

To make your `PATH` changes permanent you would add the `export` line to `~/.zshrc`.

If you change your `PATH` in the middle of a session, you should run the `rehash` command. It will rebuild the list of executables you can run to reflect the changes you've made.

3.7.4 `alias`ing commands

Many times you end up typing the same set of commands over and over. One familiar command to you should be *ssh*, which logs you in remotely to some server. At the command line you probably often type something like what's shown below:

```
% ssh username@knuth.cs.hmc.edu
```

That's a lot of typing for such a simple act. To reduce typing the shell allows you to alias a complex command and refer to it with a different name.

For example, we could alias the above command to something simpler, like `knuth`. This name has everything that you want in an alias: it's descriptive, it's short, and the name doesn't conflict with any other commands on the system. To create this alias from the command line you would type:

```
% alias knuth="ssh username@knuth.cs.hmc.edu"
```

alias uses a simple `NAME=VALUE` scheme, but when `VALUE` contains spaces it is usually quoted. In this example, the quoted portion (normal quoting rules apply) is now executed whenever you use the `knuth` command. However, this alias would only last as long as you had the shell open—generally you want to keep aliases around permanently. To keep aliases around you add them in the same form as shown above to `~/.zshrc`. Aliases in your `.zshrc` will be created each time you login. You can have as many aliases in your `.zshrc` as you would like.

Execute `alias` with no arguments to view your current aliases.

When using aliases here are some words of warning. An alias can have the same name as another command on the system. When this is the case the shell will execute the *alias* rather than the command you've typed. This can be a very bad thing. Say we didn't like the completely destructive behavior of *rm* and aliased it to *rm -i*. Then when we tried to remove files, we would have to do so interactively, but so would any programs wanting to use *rm*! This can lead to a false sense of security and is on principle a Bad Idea ©. To get around an aliased command you can always refer to it using its full pathname: `/bin/rm` instead of `rm`.

Here are some of the aliases that I have in my `.zshrc`:

```
alias ls="ls -FG"
alias knuth="ssh mkegel@knuth.cs.hmc.edu"
alias ..="cd .."
alias sizeof="du -sh"
```

3.8 Wasting Time

I figure no UNIX manual could be complete without explaining how to waste some time on the command line.

The best way I've found to waste time, if you're a reader and like quotations, is the *fortune* command. *fortune* queries a rather large database of quotations and returns one at random. You can read *fortune*s for hours. The `-o` switch provides offensive quotes that are not for a faint of heart, but they are funnier and more interesting.

It is said that *emacs* has everything, including the kitchen sink. While you can't play with the kitchen sink when you're using *emacs*, you can play tetris. The graphics suck, frankly, and music is rather lacking, but on a late Friday there's nothing better. To play tetris in *emacs* use the `M-x` shortcut and then execute `tetris`. The arrow keys control the blocks, spacebar force drops. *emacs* offers a few other games as well.

For the more adventurous types there's always *angband*. *angband* is a command line adventure and role-playing game. It is also more addicting than heroin. The game is loosely based on the Tolkien universe, and has you exploring dungeons, killing monsters, and collecting the best equipment before you finally face Morgoth. *nethack* is a very similar game to *angband*, and both are classics.

On the social front there's IRC. Before AIM the social types of the world used IRC, and if they are over 30, they still do. For those that don't know, IRC is, more

or less, multiplayer notepad. You can find an IRC that discusses quilting, Xtreme fishing (ask about the dynamite), or even interesting nth degree polynomials. The urbane FreeBSD Help group is well known for its kind and gentle nature toward inquiring new users, especially those who have trouble RTFM.

3.9 Basic Shell Scripting

Being able to write small shell scripts is one of the niftiest things you can know. Why? Well, let me ask you: "How would you rename a bunch of files from the command line?". Your first guess might be something like this:

```
% mv *.a *.b
```

This is obviously wrong; see the section on *mv* if you don't know why. However, we do know that when given just two filenames *mv* will rename the file to the second filename. The basic construct we are looking for must then be a for loop that would execute a single *mv* command on each file. The following snippet does what we want:

```
% for i in *.a
for> mv $i `basename $i .a`.b
```

Don't worry about the above snippet; it serves only to illustrate the usefulness of shell scripting. What's important to note is that we did this complex operation from the command line. Because the shell provides a Turing-complete language, we could string together commands like this to do any sort of complex operation.

Because the topic of shell scripting generally requires a book, I'm only going to skim over the basics of variables, for loops, and if statements. This should do most of what you need.

Creating shell scripts are easy, but they do require some magic. Every shell script has a sha-bang to indicate that it is a script. Python, Perl, and Ruby also use this convention. The sha-bang tells the OS what program must be run to interpret the commands and always occurs at the very beginning of the file. The following snippet illustrates many important shell programming concepts:

```
#!/bin/sh

# This is a comment.
# Now for a variable assignment:
MYVARIABLE=42

echo "This is my shell script."
echo "answer = $MYVARIABLE"
```

In this example, we are using `/bin/sh` as our shell. This is the standard UNIX shell and is what shell scripts are written in to maintain portability. Other shells

(/bin/zsh, /bin/bash) and languages offer more features, but you cannot count on them always being available. Stick with /bin/sh for now. Note: make sure the file is executable, otherwise it won't do anything.

Variables are usually given uppercase names. Notice the spacing in the assignment, this is important! Variables are treated kind of like C pointers: they must be "dereferenced" to retrieve the value, otherwise they are just a string. To dereference a variable, precede the name with a dollar sign ($). Recall that double quotes don't deactivate $, but single quotes do (and everything else for that matter). Note: variables can be undefined—dereferencing an undefined variable returns a zero-length string.

A shell script is really nothing more than a list of commands. Commands are executed line by line, so to separate them you would use a semi-colon. Commands are typed in normally, as you would on the command line. A list of commands isn't all that useful, however, unless you have some control flow mechanisms.

The simplest control flow mechanism is the if statement. Here is a basic script to see if you are root or not:

```
#!/bin/sh

# Test for string equality
if [ "$UID" == "0" ]
then
    echo "I am root. Yippee!"
else
    echo "I an not root. Much sadness."
fi
```

All if statements follow this form, though you may omit the else construct. Now for the magic: [is actually a hardlink to a program called test, and is what performs the actual test. The if..then..else..fi construct just tests the return code of [, and makes a decision based on that.

In UNIX when a program successfully returns it returns a value of 0; when unsuccessful it returns non-zero. This is why you'll see a return 0 or a return ERROR at the end of C programs. This is probably the reverse of what you are used to, but in UNIX the roles of "true" and "false" are reversed: true = 0, false != 0. You'll need to keep this in mind as you write both shell scripts and UNIX programs.

You can test a number of different things with [. The following snippet illustrates a few of the more useful ones:

```
#!/bin/sh

# Test for string equality
[ "$UID" == "0" ]

# Test for file existence
```

```
[ -e /some/file ]

# Test for non-zero length string
[ -n ‘echo $UNKNOWN_VARIABLE‘]

# String tests: equal, not-equal
[ "one" =  "one" ]
[ "one" != "two" ]

# Integer tests: equal, not-equal, greater than, less than
[ 2 -eq 2 ]
[ 1 -ne 3 ]
[ 3 -gt 1 ]
[ 5 -lt 9 ]
```

Expressions can be combined with the `-a` switch (AND operator) or the `-o` switch (OR operator). A full listing of available switches can be found by running `man test`.

You've already seen a basic for loop. The basic syntax of a for loop is this:

```
for arg in list
do
    something cool
done
```

The `list` can either be a fileglob (`*.txt`) or a set of strings (e.g. "Mars" "Venus" "Earth"). The following examples illustrate both options:

```
#!/bin/sh

for arg1 in *.txt
do
    echo "$arg1 is a text file"
done

for arg2 in "Mars" "Venus" "Earth"
do
    echo "$arg2 is a planet"
done
```

That's basically it for for-loops.

There are some other miscellaneous things to know, however. Arguments can be passed in to a shell script. Assume we have a script called `extinguish.sh` that needs to take the name of a printer and put the fire out. You would call the script like this:

```
% extinguish.sh gute.cs.hmc.edu
Checking for gute.cs.hmc.edu
Printer is on fire. Putting fire out.
```

The variable `$1` holds the first argument, in this case `gute.cs.hmc.edu`, `$2` holds the second argument, and so on. The variable `$0` holds the name the script was invoked as, here `extinguish.sh`.

One last thing, to access the return code of the last run command you use another special variable: `$?`. This is often when you need to know whether a program was successful and not if it just printed lots of output.

For more information about shell scripting see these references:

- Advanced Bash-Scripting Guide
- Learning the bash Shell (O'Reilly)
- Wicked Cool Shell Scripts (No Starch Press)

3.10 Other UNIX Shells

UNIX by its very nature generally offers more than one way to do things. Shells, for instance, are no exception. By default new student accounts starting in the fall of 2005 are being setup to use the Z Shell, or *zsh*. The Z Shell is probably the best shell in widespread use and offers many excellent features that you can't find in other shells. For this reason we would suggest that you stay with *zsh*, and that you explore what it has to offer. We think that you'll be impressed.

If you aren't deterred, however, and are a little curious about what other shells are out there in the UNIX badlands then come listen. The most widespread shell is the Bourne-Again Shell, or *bash*. This shell is a re-implementation of the original UNIX Shell, *sh*, created by Steve Bourne. It has since amassed a number of features and continues under active development by the Free Software Foundation. As of this writing *bash* version 3.0 had been released. One important thing to note: most UNIX systems do not come with *sh*, instead *sh* is a hard link to *bash*. Learning to use *bash* can be valuable, both for scripting purposes and for when you use systems that do not have *zsh* installed.

The other really popular shell is *tcsh* (T Shell). Before we switched to *zsh*, students by default used *tcsh*. This shell is an extension of the C Shell, which was the first *user* oriented shell, and has things like history editing and tab completion. This won't be very exciting to a *zsh* user, but in 1987 it was pretty cool.

There are countless other shells if you go digging on the internet, some of which are nothing more than hobby projects. Others are very cool and may someday overtake the current batch of shells. In any case, try things until you find something that fits or if nothing fits go make your own. That's the UNIX way!

3.11 Personal Webserver

If you have previously done web development you'll be pleased to know that you can take advantage of the CS department webserver. Placed in each student's home directory is the directory `public_html`. Files placed inside this directory will be served by the CS department's webserver, muddcs. You can reach your personal website by visiting: `www.cs.hmc.edu/~username`. Note that files will need to be world readable (`o+r` permission) before anyone can read them.

If you haven't done web development before, you can set up a simple home page for yourself by creating the file `index.html` inside `public_html`. Put the following material inside `index.html` to get a basic (and I mean *basic*) webpage:

```
<html>
Welcome to _____________'s Home Page. <br>

You can find links to my hobbies, homework, and favorite porn
sites in the links below: <br>

</html>
```

3.12 Other Useful Programs

Below is a list of useful UNIX programs, all of which run under X11. Unfortunately, most of these are not installed on the CS department machines, and only a few have been ported to run under OSX. You can find and run all of them, however, with Linux. Each is an excellent application and well worth using if you are looking for an open source (and free!) alternative.

web browser	firefox, opera, konqueror
text editing	jedit, emacs, vi
word processing	abiword, open office, koffice
typesetting	latex (this book was typeset with LaTeX)
spreadsheet	gnumeric, open office, koffice
music	xmms, amarok (my favorite)
movies	totem, xine, mplayer, vlc
terminal	eterm, aterm, xterm, rxvt, gnome-terminal, konsole
mail	thunderbird, evolution, pine/mutt through xterm
pdf viewers	xpdf, kpdf, gpdf
ftp	ncftp or lftp (cmd line), gftp
cd burning	k3b
aim	gaim, kopete
bit torrent	azureus
file manager	konqueror, nautilus
spell checkers	ispell and aspell

Chapter 4

Programming at HMC

The previous chapters have tried to help you get acquainted with the shell and some of what's possible on the command line. But most of your time in `CS*` will not be spent grepping through text files or looking at manpages. This chapter is here to help you get started programming and show you what tools you'll be using, what tools you'll want to learn, and some tools you've never even heard of.

4.1 Text Editors

The most basic tool of a programmer is his office chair and coffee mug. But there are times when the pointy-haired professor will expect some productivity and you won't have an aunt on her deathbed or a cousin getting out of prison to get you away from school for two to six weeks. At times like these real Mudd programmers buckle down and surf the web looking for a better text editor.

The text editor you work in is just as important as the environment outside your computer. Just as some people are distracted by loud music, friends next door, or an interesting book to read, a poor text editor can make you that much less productive. A good text editor won't make you a great coder, or even a good coder, but it can help make your code cleaner, clearer, and easier to read if to no one other than yourself. **Learn to use at least one good text editor.**

You should probably look for a text editor with the following features:

- **Syntax Highlighting:** Syntax highlighting shows the structure of programming expressions either through color or font typesetting or a combination of the two. For example, keywords may be highlighted in green, while numeric constants are italicized and highlighted in red. This can be an unbelievably helpful feature when visually scanning through code.
- **Low eye-strain color scheme:** Sitting and staring at a computer monitor all day can be really stressful on your eyes. To combat this try to use a color scheme that isn't quite so bright, like white text on a black background. While this is more a general tip, make sure that you can easily change your

text editors default color and font values to something that fits your needs.

- **Low eye-strain font:** Having a good font is not nearly as important as having a good color scheme. Nevertheless, a font which makes you work to discern every character on the screen is probably not a good choice. Play around until you find something suitable. Also, remember that you'll want to stick with monospaced fonts.
- **Line numbering:** Almost every compiler error will include a line number allegedly showing you where the error occurred. You need to be able to quickly find these lines, otherwise debugging will be an unusually cruel variety of hell for you.
- **Column numbering:** You should always keep lines to 80 characters in length. This is sound advice for when you take CS70, but since we don't live in 1984 anymore I wouldn't take it too seriously after that. Screen real estate is much more plentiful, and lines up to 100 characters are very reasonable. Just don't go overboard on this. Code is meant to be readable and long lines inhibit easy reading. Balance is key.
- **Multiple Views:** Having a text editor that lets you work on multiple files at once can be invaluable. While sometimes a function of screen size (we don't all have a 24" inch Widescreen Dell) having multiple views shouldn't be something you sacrifice in a text editor.
- **Soft Tabs:** When working with plain text try to avoid using actual tab characters like the plague. Use spaces whenever possible and enable soft tabs for your editor. When using soft tabs spaces are inserted, instead of a tab character, when you hit the tab key. Tab characters are and should *always* be interpreted as being 8 spaces, but soft tabs can be as many spaces as you would like. I use 4-space soft tabs. Soft tabs help in two ways: code is displayed the same in any and every text editor, and you can save on line length when indenting. Eighty character lines can be more than enough with soft tabs.
- **Custom Keyboard Shortcuts:** Being able to change the default keyboard shortcuts can be extremely valuable, especially if your editor allows some complex operations from the keyboard. Ergonomics may also suggest a different layout than the default.
- **Secure FTP Support:** SFTP allows you to login and remotely edit documents. This is a rather obscure feature, and is generally not present in most text editors where editing is assumed to be done locally. At Mudd, however, STFP support can be essential if for no other reason than convenience. SFTP does not have to be built-in to your text editor—anything that allows you to mount your home directory, as either a network share or another hard drive, would work just as well. For editors that work through the console, like *emacs*, having SFTP support is unimportant.
- **Extensibility:** Having the ability to script actions is key to the longevity and usefulness of an editor. One of the principle reasons that *emacs* has survived for so long is that users can extend it by adding new functionality. Scripts allow you to generally reprogram your editor with new language support,

keystroke actions, utility support, etc. Newer editors feature a plugin system for this exact reason.

Don't take any of the above features as a must have, for example some people, maybe even yourself, find syntax highlighting to be distracting and would rather not have it. The trick is to find an editor that works for you and lets you be productive without being frustrating to use. To help you out I've listed some of my favorite text editors in the sections below. Each editor below meets most, if not all, of the features described above. I've given *emacs* a section of its own since knowing how to properly use *emacs* can save you hours of frustration. Also, each editor discussed below can be found on the computers in the Terminal Room.

4.1.1 A Few Good Choices

- `jEdit`: jEdit is a great general text editor. Because it's written in Java, it works on a variety of systems, including Windows, Linux, and OSX. jEdit itself was written by and for programmers, like *emacs*, so it provides a well honed programming environment. It is also highly customizable and extendable through the use of plugins. One notable plugin provides SFTP support for remote editing.
- `TextMate`: TextMate is an excellent OSX text editor that the CS department offers on all the Macs. While TextMate is a programmer's text editor, it is not as inherently powerful as jEdit or emacs. Despite this, TextMate still offers a very comfortable programming environment. Two nice features are in-line spell checking and a specialized form of tab completion.
- `SubEthaEdit`: SubEthaEdit is an alternative OSX editor on the CS department Macs. The claim to fame of this text editor is that it offers collaborative editing, but it offers little beyond this. With collaborative editing two or more people can edit the same document at the same time while working at separate computers. While not allowed for either CS60 or CS70, it is still a nifty feature.
- `vi`: *vi* is a modal editor, which is confusing for most new users. Once a user has learned the basics of *vi*, however, they can usually do quite complex editing functions in just a few keystrokes. Beware that while *vi* is quite powerful, it is not for the faint of heart or those who have better things to do with their time. *vi* has spawned many clones, most of which improve on the original. The most famous of these is *vim*, or *vi* improved. There is also *gvim*, a graphical version of *vim*, and *vile*, an *emacs*-like *vi* for those with identity issues.
- `ed`: The pinnacle of editor minimalism and simplicity wrapped in the clean 1970s lacquer veneer of a command line interface. If you like *vim*, you'll love *ed*.
- `pico`: The UNIX command-line equivalent to Windows' Notepad.
- `emacs`: see the section below.

4.1.2 What To Use In Case of Emergency: `emacs`

In case of emergency you can always just use *emacs*. But don't brush off *emacs* too lightly. *emacs* is quite possibly the most versatile and powerful text editor that you can use. Because it has been around for over two decades, *emacs* is battle tested and has found itself ported to many different platforms. There are even native, if not up-to-date, versions for OSX and Windows.

Inevitably you will find yourself needing more than what your current editor offers. The ability to customize *emacs* makes it perfect for those users who desire either to customize, or even completely extend, their environment.

Now for the shocking news: by default *emacs* runs from the command line. No GUI, no shiny buttons, no annoying piece of spirally twisted wire asking you what you're writing, just a plain, simple interface. This actually makes *emacs* very useful to learn, since there will be times when all you have is a command line.

But don't get down too much, *emacs* isn't command line only. It does support, when available, an X11 GUI and the native OSX and Windows versions both use a GUI. Newer versions support a mouse wheel, and overall GUI support is excellent.

The following resources should prove invaluable when learning *emacs*:

- `www.gnu.emacs.org`
- Learning GNU Emacs (O'Reilly)

should put in more about emacs - commands, how to modify (.emacs - include sample), etc.

4.2 Working From Windows

Putty

WinScp

Cygwin

include graphics - would help a lot

Doing your homework from Windows presents one of two problems: (1) you can create, edit, and debug your homework files locally, then transfer them to knuth; or (2) you can edit your files remotely working exclusively on knuth. In either case, your homework needs to end up on knuth in a usable state.

The first option, unfortunately, isn't really much of an option at all. That isn't to say that you can't do your Java, C, or C++ homework on your local Windows box, but rather that you'll have fewer problems if you work exclusively on knuth. There are two reasons why this is the case. The first reason is that while there is a UNIX-like environment, Cygwin, things don't always compile correctly under Linux even though they may have been fine under Cygwin/Windows. Your files need to compile cleanly under Linux; anything else is irrelevant. The other reason is that Windows uses a different line ending scheme than UNIX. Therefore, you cannot just blindly copy files; they have to be converted. This is not a big issue, but not converting makes your code harder to read and you will lose points.

If you are up to the hassle of editing your files from Windows, you do have a couple of options. You can mount your knuth home directory as a network share, using a protocol like SFTP. Or, use a text editor, like jEdit, that has an SFTP plugin so that you can edit files remotely. An excellent native Windows editor with SFTP support is Crimson Edit. With either option you should be able to do your homework from the comfort of your room and only a minimum of hassle.

4.3 Working From Mac OS X

include graphics here also

Terminal.app is really all that's needed

4.4 Submitting Homework

In other courses you usually have written homework, but in CS courses almost all of your work will be electronic. This is especially true in CS60 and CS70. Because of this, you will be required to electronically submit your homework. Submission is handled by the *csXXXsubmit* program, where XXX is the course number. While taking CS60 you will be using *cs60submit* to submit your homework files. You can submit as many times as you'd like; in fact this practice is recommended so that graders and professors can see your progress on the assignment.

csXXXsubmit tends to be fairly intelligent about which files you can submit. For example, *cs60submit* will only allow you to submit files with the following extensions: isc, rex, java, html, pl, txt, FA, TM, and jff. You should never need to submit files while taking CS60 with an extension other than those given.

csXXXsubmit follows standard UNIX convention whereby files to submit are given as arguments on the command line. When run without any arguments *cs60submit* prints a usage statement. The following example shows how to submit your homework, assuming its all in Java:

```
% cs60submit *.java

I need to know a few things before I submit your homework...

What assignment are you submitting?
```

You will be required to enter which assignment the files are for, after which the filenames of each file submitted will be printed.

Once your files are submitted, *csXXXsubmit* will email you copies of each of the files you have submitted. This is an archaic hold over from times past, but it does allow you to keep a record of what you've submitted and when. Watch out, however, as this can quickly fill your quota if you constantly submit large files.

Due to limitations with the current submission system, CS70 is using a different set of programs: *cs70checkout*, *cs70checkin*, *cs70resync*, and *cs70submit*. Information about these programs will be provided to you by your CS70 instructor.

4.5 CS60: An Introduction To Annoying Languages

Getting through CS60 requires only a basic understanding of UNIX commands. While you may not be using `sed` and `find` every day, you will be writing code every day. Currently there are three major languages taught in CS60. The following sections cover how you compile code in each of these three languages.

4.5.1 `Java`

Java is an architecture independent language originally developed by Sun. Java is a nice introduction to the power of C and C++, without many of the hassles of these two languages.

Let's assume that you've written program that prints something to the screen in Java. This program is implemented in the file `vexed.java`. To compile this file you use the `javac` command. This is the Java Compiler and is what turns your statements into Java byte-code. Let's do this now:

```
% javac vexed.java
```

In true UNIX fashion, no output is good output. If you have errors they will be printed to the terminal. Obviously if you have many errors you will want to pipe them into a pager, like *less*, to make viewing them easier.

Once compiled you should see a file called `vexed.class`. This is the compiled version of your `hello` class and is what will actually be executed. To execute your program you use the `java` command. This command invokes the JavaVM, which is what executes your program. Let's do this now:

```
% java vexed
Kill Me Now.
```

Notice that you specified only the class name and not the full filename. If you specify the full filename you will get an error, since *java* interprets things a bit differently. In Java projects that contain multiple classes, you will want to execute the one containing your `main()` function.

4.5.2 `Rex`

Rex is a home-grown interpreted functional programming language. You invoke the *rex* interpreter with the `rex` command. Unlike programming in Java, where you have a main loop that is invoked at runtime, *rex* has no such functionality. Instead, you write functions that can be invoked interactively from the interpreter. Python users should be at home using *rex*. You exit the interpreter with `C-d`.

As an example of using *rex*, say we have a file named `irk.rex` that defines functions: `fire` and `brimstone`. The following snippet shows how to have *rex* load `irk.rex`, and some interaction:

```
% rex irk.rex
irk.rex loaded
1 rex > fire("really hot");
Ouch, that burns!

2 rex > brimstone(fire("really really hot"));
And you thought Java was hell?

3 rex >
```

Additional filenames may be specified so that multiple *rex* files may be loaded. If no filenames are given the interpreter will start with a blank environment. *rex* has a builtin help function that can be accessed with the `*h` command. See the Rex Reference Card on Professor Keller's homepage for more information about using Rex.

4.5.3 `Prolog`

Prolog is probably the most unique language you are likely to encounter during your tenure at Mudd. Prolog is a goal-oriented language. However, it is probably better described as being a logician's wet dream, but don't let this image sear your subconscious.

Assuming you've written some Prolog and actually want to run the code, you can invoke the Prolog interpreter with the `prolog` command. The following example shows how to run Prolog code contained in the file `rankle.pl`:

```
% prolog rankle.pl
Yes.
No.
Maybe.
What was the question again?
Do I get a cookie?
No.
```

Like Rex, running `prolog` without any arguments will invoke the interpreter with an empty environment. More information about Prolog can be found on the CS60 website or by interrogating Professor Keller.

4.6 CS70: Trial By Firing Squad

Don't let the reputation of CS70 scare you. It will leave a few psychological scars, but in general we prefer to mark CS majors with branding irons. Oh! The sweet, sweet smell of burning flesh after grading CS70 assignments.

CS60 is pretty light on testing your UNIX knowledge; CS70 is very different. Part of the purpose of CS70 is to teach you how to create good UNIX programs and what facilities you need to know to write, compile, and test your programs.

The following sections outline the basics of what you need to know in CS70.

4.6.1 Compiling With `gcc` and `g++`

While taking CS70 you will become intimately familiar with GCC, the GNU Compiler Collection. GCC includes support for compiling dozens of languages, but you should be concerned with only two: C and C++. Besides random switches the only thing you need to remember is that code written in C is compiled with *gcc*; code written in C++ is compiled with *g++*.

There are more than a hundred different command line options that both *gcc* and *g++* accept; I will only cover a few of the more basic ones. The switches discussed below work in either program.

Assume you are trying to compile a simple C program, contained entirely in one file: `myprogram.c`. The basic syntax of *gcc* is quite simple:

```
% gcc FLAGS FILES -o EXECUTABLE_NAME
```

FLAGS may be omitted, though there are some you are advised to use. FILES are those source code files you want to compile. The `-o` switch lets you name the executable *gcc* will output. The default name is `a.out`.

Here is how you would want to compile and execute `myprogram.c`:

```
% gcc -g -Wall -W -pedantic myprogram.c -o myprogram
% ./myprogram
```

While it is possible to compile programs that are composed of more that one source file from the command line, it is generally not good practice to do so. This is why *make* and `Makefiles` were invented.

Good programming practice dictates that you should always use certain switches. You are well advised to use the first four switches in the table below.

`-g`	Builds in *gdb* debugging symbols
`-pedantic`	Output all errors violating the ISO C and C++ standards
`-Wall`	Output all compiler warnings
`-W`	What does this do?
`-Werror`	Treat all warnings as errors
`-O#`	Code optimization level; # can be 0, 1, 2, or 3
`-l`*lib*	Link with shared library *lib*
`-L`*dir*	Link with objects in *dir*
`-I`*dir*	Look for header files in *dir*
`-o` *name*	Output the compiled execute with name *name*

4.6.2 `make`ing Your Programs Work

When compiling source code *gcc* first makes object files. These files have the extension `.o` and contain machine code that has yet to be linked. Each `.c` file you write will be compiled into a `.o`. These files are then linked together to create the final executable. Thus, when changes are made to your source, only a few new object files (presumably) and a new executable need to be created.

make (default rule), make all, make install, make depend, make clean
make file template
make alternatives - ant (java/XML), rake (ruby), scons (python)

4.6.3 Debugging With `gdb` and `ddd`

4.6.4 Code Profiling

4.6.5 trace and truss

do i really need this section?

4.7 Version Control With Subversion

Imagine for a moment that you are working on a large project, with perhaps a hundred files, and hundred's of thousand of lines of source code. Or it could even be a small project, ten files with a couple thousand lines. In either case a bug gets introduced that you *know* was not there last week. How do you rewind time and get back what your project was like last week? Or what if you accidentally delete or corrupt a file? These are some of the many problems that version control helps you resolve.

The basic idea of version control is to take snapshots of a project, whatever that project may be. It doesn't have to be a coding project even, for example the code for this book is under version control. These snapshots are then stored in an archive somewhere and can be queried against or downloaded in full. They provide a *history* of how your project developed. They can also save your butt.

Version control is actually quite old and there have been several utilities created to perform this function, like RCS (Revision Control System), CVS (Concurrent Versions System), and Subversion. RCS is ancient, but is still used in some contexts, like professors who are too stubborn to change. CVS is newer and is quite popular among Open Source projects. The one that will be discussed here, however, is Subversion.

Subversion, while not perfect, is light-years ahead of either RCS or CVS. It is incredibly simple to use and creating a repository could not be easier. And the documentation is excellent, so you shouldn't have any problem finding out more.

To use Subversion from the command line you use the *svn* program. There is also the *svnadmin* program which handles repository creation and other administrative tasks, but for the most part you'll just be using *svn*.

4.7.1 Checking Out The Repository

All version control systems have some sort of archive and Subversion is no different. In Subversion terms this archive is known as a repository. To setup a repository you first need to create a suitable directory hierarchy. I have a root directory and then subdirectories containing the repositories for each project I have. You'll generally want to keep one project per repository, but there is nothing in Subversion to keep you from doing otherwise. Listed below is an example hierarchy:

```
dir/to/repository/
                 |----cs60/
                 |        |----hw01/
                 |        |----hw02/
                 |----cs70/
                 |        |----chunkystring/
                 |----unix_course/
                 |----unix_book/
```

Say we want to create a repository for our `chunkystring` assignment. Assuming that we created what we have above, we would then navigate to the `chunkstring` directory. To actually create the repository we would then execute the following command:

```
% svnadmin create
```

That's it! You now have a standard Subversion repository. You can of course change how the repository works, like restricting access, but that's beyond the scope of this manual.

Once the repository is created you can then start to use it. The first thing that you need to do is "checkout" the repository. You only have to checkout the repository once. To checkout the repository for `chunkystring` you would issue the following command, which will be explained below:

```
% svn checkout \
> svn+ssh://USER@SERVER/dir/to/repository/cs70/chunkystring ./
```

svn is the command line client for Subversion and is what you'll use to manage your project. *svn* takes a number of different options: the `checkout` option checks out a Subversion repository from some location to the location specified. The long second argument specifies the full URL and pathname to where the repository is that you want to checkout. It can either be on the local machine or on some server. In the above example you are checking out the repository over some network, which is the most common case. This example is discussed in more detail below.

The first thing specified is the URL protocol: `svn+ssh://`. Subversion supports a number of different protocols, this one being useful when you want restricted repository access, yet still network accessible. To access a repository on your local machine you would likely use the `file://` protocol. Other supported protocols include: `svn://`, `http://`, and `https://`.

Getting back, the above example will login with *ssh* and download the repository from the server `SERVER`, with `USER` as your username. After specifying the server, the directory path to the repository is given. Most servers will have a sane pathname for their repositories. The last argument given is where to put the checked out repository. In this case it is being checked out to `./`, or the current directory. Read the documentation if you get any errors.

4.7.2 Working With Files

Now that you have the repository checked out (the hard part) you need to start using it (the annoying part). When you checkout a repository you are said to have a "working copy" of the repository. All subsequent actions are done to this working copy. The first thing that you'll need to do is add all the files and directories that you want under version control to the repository.

Note: Each directory under version control contains a `.svn` subdirectory that records what files are under version control, what version they are, etc. DO NOT PLAY WITH OR DELETE THIS DIRECTORY UNLESS YOU KNOW EXACTLY WHAT YOU ARE DOING.

Generally you'll want all of your files under version control, but you can specify them individually or with a file glob. The following example adds some files to a repository:

```
% svn add chunky.cc chunky.h
```

Similarly there are times when you want to remove files from version control. The following example removes and deletes the two files that were just added:

```
% svn delete chunky.cc chunky.h
```

You can also copy and move files or directories, like so:

```
% svn copy chunky.cc chunky_two.cc
% svn move chunky.h chunky_header.h
```

The new file `chunk_two.cc` is now under version control and contains a copy of `chunky.cc`.

The Subversion copy and move use the same semantics as *cp* and *mv*, so you can specify a file list and then copy/move files to that directory. DO NOT use the regular *cp*, *mv*, and *rm* commands, since both Subversion and you are likely to get confused. Use the commands builtin to *svn* for these tasks instead.

4.7.3 Updating the Working Copy

Subversion has been designed to be a multiuser system. Any number of people can checkout a repository and work on it alone, making whatever changes they may. This means that you need to keep your working copy up-to-date. You do this using the `update` argument to programsvn, like so:

```
% svn update
A  chunky_three.cc
D  test.c
U  chunky_header.h
C  chunky_two.cc
G  chunky.cc
```

This will bring your copy up to the newest revision. The letters `A,D,U,C,G` followed by a filename indicate something special about the file listed. Below is a table of what each letter means:

A	Added
D	Deleted
U	Updated
C	Conflict
G	Merged

The only one to really pay attention to are those files that have conflicts. Before you can commit any changes you have made you need to resolve all conflicts. Merging conflicts by hand can be intimidating the first time, but after a while should be as easy as skydiving (you only mess up once!).

4.7.4 Play Nice Children: Resolving Conflicts

A conflict within a file has two parts: what you have, and what the updated copy has. Below is an example of what a conflict might look like. Suppose you and Meghan have a list of what liquor you need to restock your bar, but there is some miscommunication. Here is a printout of the file `liquor_to_buy.txt`:

```
% cat liquor_to_buy.txt
Maker's Mark
Captain Morgen
Grey Goose Vodka
Chopin Vodka
Malibu
<<<<<<< .mine
Bombay Sapphire
Kahlua
Blue Curacao
=======
Amaretto
Peach Schnapps
Midori
>>>>>>> .r2
Jack Daniels
Bacardi Rum
Jose Cuervo
```

As you can see from the file you disagree on what needs to be bought. The conflict is denoted as text between <<<<<<< and >>>>>>>. What you had was this:

```
<<<<<<< .mine
Bombay Sapphire
Kahlua
Blue Curacao
=======
```

The ======= acts as a separator between the conflicting parts. The text in the new revision comes after the separator and is this:

```
=======
Amaretto
Peach Schnapps
Midori
>>>>>>> .r2
```

You resolve the conflict by merging the two pieces of text. Here is an example of what the two parts should be merged to:

```
Kahlua
Blue Curacao
Midori
```

Having resolved the conflict `liquor_to_buy.txt` would then contain the following:

```
% cat liquor_to_buy.txt
Maker's Mark
Captain Morgen
Grey Goose Vodka
Chopin Vodka
Malibu
Kahlua
Blue Curacao
Midori
Jack Daniels
Bacardi Rum
Jose Cuervo
```

Once you have resolved all conflicts you need to tell Subversion. You do this using the `resolved` argument. We would resolve the conflict on `liquor_to_buy.txt` like this:

```
% svn resolved liquor_to_buy.txt
```

Sometimes conflicts can be so hairy that there is no way to easily resolve them. In these situations you can just remove the file using *rm* then run `svn update` again. The file will be replaced with the old copy and you can then run `svn resolved` on the file. This is not a recommended practice, but works nonetheless.

4.7.5 Committing Changes

To commit your changes you use the `commit` argument like so:

```
% svn commit
```

You will then be asked for a log message. The log message should describe what it was that was altered/fixed/broken/etc. Try your best to be descriptive in your log messages, since you may some day need them and have no one to blame but yourself if they are useless.

You can commit or update from *any directory* you have under version control. The directory, and its subdirectories, that you update/commit from will then be updated/committed without touching any of the other directories in the hierarchy. This is useful if you keep getting errors in one directory, but need to be working in another.

Note: Subversion doesn't by default know which editor to use so it checks the `VISUAL` environment variable and then `EDITOR`. If neither are set, then you will get an error. Set either to your editor of choice if you are getting this error. See section 3.7.1 for more information about Environment Variables.

4.7.6 Viewing Your Log History and Status

To see all of the log messages for each revision you use the `log` argument, like this:

```
% svn log
...Lots of Log Output...
```

Log messages can be useful for pinpointing which revisions have which changes and can be a lifesaver.

To see the status of your working copy files and directory you use the `status` argument, like this:

```
% svn status
...Lots of Status Output...
```

This can tell you which files still have conflicts, which files are under version control and which are not, whether a file was merged or updated in the last `update`, and more.

4.7.7 Resolving Errors

The easiest way to resolve errors is to run

```
% svn cleanup
```

This removes (or tries to remove) all the lock files in each `.svn` directory and does some other bookkeeping. It generally is not simple to resolve errors, and it cannot be easily explained here. After using Subversion for a while, though, you should be able to get a feel for how it works and fix things yourself. Of course you can always ask for help or just checkout an old copy of the repository...

4.7.8 Finding Help and Other Information

Subversion has a fairly extensive builtin help system. To get general help with *svn* you use the `help` argument, like this:

```
% svn help
```

You can also get help for each Subversion command. Below is the general syntax for getting help with a command `COMMAND`:

```
% svn help COMMAND
```

The ultimate resource is the Subversion book: *Version Control with Subversion*, published by O'Reilly. You can find a free, online copy at: `http://svnbook.red-bean.com/`. This book explains everything that you need to know about using Subversion, and even some stuff that you don't.

There is one last piece of information. The CS Department has a server, `babar.cs.hmc.edu`, whose only job is to act as a Subversion server. You can inquire about using this server by asking Staff.

4.7.9 Table Of Commands

Subversion is very user-friendly and allows a number of abbreviations for common commands. In the sections above I have used the full command name, in this table, however, I have instead chosen to use the available abbreviated versions. These commands are intended to be used with the Subversion client, *svn*.

`co svn+ssh://USR@SVR/RPATH .`	checkout a working copy to `./`
`co file:///RPATH .`	checkout a working copy to `./`
`add FILE ...`	put FILE(s) under version control
`rm FILE ...`	delete FILE(s) from repository
`cp FILE1 FILE2`	copy contents of FILE1 to FILE2
`cp FILE ... DIR`	copy FILE(s) to directory DIR
`mv FILE1 FILE2`	rename FILE1 to FILE2
`mv FILE ... DIR`	move FILE(s) to directory DIR
`up`	update the working copy
`resolved FILE ...`	resolve conflict in FILE(s)
`ci`	commit changes to repository
`st`	print the status of working copy
`log`	print the log messages for each revision
`cleanup`	hopefully fix errors if you have any

4.8 The Well-Written Program: Coding Style

what cs70 graders will look for and how you ought to start coding in cs60

indenting - make code readable indenting styles - be consistent

comments - explain for everyone
write for correctness, then optimize
good coding practice
how to debug

4.8.1 Good Code

significant examples of good code and bad code example's need to cover a variety of situations

4.8.2 Bad Code

4.9 UNIX Programming

4.9.1 Anatomy of a "Good" Program

4.9.2 Philosophy

should geoff write this and the "good coding style" sections?
one program, one purpose
KISS, POLA, etc.
formatting arguments
how to take input, format output
further resources: "art of unix programming"
section should be both about how to write good UNIX programs, and how to write programs that work well in general

4.10 Point and Spray: Debugging

preventative programming
printf
gdb/ddd
assert statements

4.11 UNIX Internals

how processes run
fork/exec model of computation
signals
threads
sockets
virtual memory - malloc, sbrk, free