Using Unix in the QED

1998 Sean Parkinson

Contents

Chapter 1

Introduction

1.1 Forward

This document is part of a program being developed to meet the needs of both new and experienced computing resource users at the Queen's Economics Department (QED), including students and faculty. It is designed to meet the specific demands of users within the QED by introducing programs and systems used widely, such as LaTeX and the Unix environment. The program also includes a group of computing advisors available to answer questions and a collection of related documentation that should be available later this year.

We hope that this project will open a new world of information and resources to new computer users as well as help experienced users to further explore the resources available. This document, like the project, is still under development. If you have any ideas for something to be added, removed or altered, please let me know.

Sean Parkinson <parkinss@qed.econ.queensu.ca>

1.2 Organization of this Document

Using Unix in the QED is a guide to some of the computing services offered in the Department of Economics at Queen's University. This document is organized into ten chapters. Listed below are brief descriptions of the information you will find in each of the chapters.

Chapter 2 *Getting Started*
: It is very important that you realize that you are not an anonymous user. Your userid combined with the name of the Unix machine you use *uniquely* identifies you around the world. To protect both your Unix account and your reputation, your password should be kept a secret. Chapter 2 introduces the terms `userid` and `passwd` and how to use them.

Chapter 3 *The Unix Filesystem*
: This chapter provides an introduction to using basic Unix commands for organizing files and

directories. You will learn how to add, remove, copy, rename, compress, and print files, view directories which hold your files and more. You can also learn more advanced commands and concepts related to controlling the access other users have to your data and using MS-DOS floppy disks with Unix.

Chapter 4 *Working with Unix*
In getting beyond the basics of Chapter 3 you will learn about the Unix shell including topics such as: wildcards, pipes, redirection, job control and shell customization.

Chapter 5 *The X Window System*
Most Unix systems provide the user with a windowed environment based on X-Windows. The ability to open and use multiple windows, menus, and icons is convenient and can be used to simplify the working environment. X-Windows is similar in appearance to Microsoft Windows or the Mac OS. In X-Windows, each Unix command-line shell is contained in a separate window. This means you can use multiple Unix shells at the same time simply by opening multiple shell windows. Software designed to run under X-Windows can also take advantage of user-interface features such as menus and dialog boxes. The X-Windows chapter gives a brief introduction to X-Windows by explaining a window's components and operations.

Chapter 6 *File Editors*
Many things you do on the computer require the ability to edit text files. Examples include composing the body of an e-mail message, writing HTML for web pages you wish to publish, or writing code for programs you are creating. *File Editors* introduces the editors Emacs and pico. Emacs, the focus of this chapter, is a very large program that integrates many of Unix's tools into one interface. Emacs performs complicated editing operations on your files and allows advanced users to customize and extend its functionality. Pico, although less functional–is much faster and easier to use than Emacs. *File Editors* explains the basic commands which allow you to add, edit and delete text, perform cut, copy and paste operations, undo changes, and save files in Emacs. The chapter also discusses search and replace commands and the on-line help, editing modes, and multiple buffers features in Emacs.

Chapter 7 *LaTeX*
Although text editors can be used to create and modify text files, they are not well-suited for formatting text. For this task we use LaTeX, a popular typesetting program. LaTeX was originally developed for Unix machines; because of its popularity, versions are now available for Windows-based PCs and the Apple Macintosh. LaTeX is not a word processor; you must first create a text file which contains both the body of your document and special formatting codes which tell LaTeX how to format the document's text. LaTeX reads your file, interprets the formatting codes, and produces a final formatted result. The LaTeX chapter here explains the 'bare-bones' commands used to generate well-organized and attractive looking documents.

Chapter 8 *Electronic Mail*
One of the most useful features of the Internet is the ability to send and receive e-mail. E-mail is quick and reliable. Moreover, e-mail messages can be sent to anyone anywhere in the world, as long as they also have an e-mail account. *Electronic Mail* teaches you how to use the e-mail program Pine.

Chapter 9 *Using Remote Systems*
Most Unix machines are connected to the Internet. Once you've connected to the Internet, you'll need to know how to access the services and information stored there. Using Remote Systems describes four Internet software packages: Telnet, FTP, ssh and scp. Telnet and ssh are used to login to remote accounts from your computer. FTP and scp are used to transfer files between your computer and remote FTP sites around the world. This chapter explains how to use these programs from Unix based machines.

Chapter 10 *Econometric Software*
This chapter is devoted to a very brief introduction to using the programs SHAZAM, TSP, Stata, Maple and LIMDEP on qed.

1.3 Conventions

These are some of the typographical conventions used in this book.

Bold Used to mark **new concepts**, **WARNINGS**, and **keywords** in a language.

italics Used for *emphasis* in text.

slanted Used to mark **meta-variables** in the text, especially in representations of the command line. For example, "`ls -l` *foo*" where *foo* would "stand for" a filename, such as `/bin/cp`.

`Typewriter` Used to represent screen interaction.

Also used for code examples, whether it is "C" code, a shell script, or something else, and to display general files, such as configuration files. When necessary for clarity's sake, these examples or figures will be enclosed in thin boxes.

Key Represents a key to press. You will often see it in this form: "Press return to continue."

◇ A diamond in the margin, like a black diamond on a ski hill, marks "danger" or "caution." Read paragraphs marked this way carefully.

Chapter 2

Getting Started

You may have previous experience with MS-DOS or other single user operating systems, such as Windows or the Macintosh. In these operating systems, you didn't have to identify yourself to the computer before using it; it was assumed that you were the only user of the system and could access everything. Well, Unix is a multi-user operating system—not only can more than one person use it at a time, different people are treated differently.

To tell people apart, Unix needs a user to identify him or herself by a process called **logging in**.

2.1 Logging In

The **login** is Unix's way of knowing that users are authorized to use the system. It also lets each user define their own personalized work environment and even work on the same computer at the same time. To log into a computer you require a userid and password. A userid is your identification and a password ensures that only you can use it.

At login[1] you should see something like the following (the first line could be anything):

```
i586-unknown-linux [SSL] (qed) (ttyp1)

qed login:
```

This is your invitation to login. Throughout this manual, we'll be using the userid `parkinss`. Whenever you see `parkinss`, you should be substituting your own account name.

`qed` is, by the way, the "name" of the machine I'm working on. There are several other Unix machines in the economics department. Those available for public use are Cox, Wald, Frisch, Waugh, Lovell, Sargan, Edith and Laffer.

After entering `parkinss` and pressing return, I'm faced with the following:

[1] Since `qed` is locked away you must connect to it over the network and log in. You can use the telnet utility or, preferably, the ssh utility to interact with `qed`. These will be discussed in chapter 9

```
qed login: parkinss
Password:
```

What Unix is asking for is my **password**. In order to promote confidentiality, when you type in your password, you won't be able to see what you type. Type carefully: it is possible to delete, but you won't be able to see what you are editing. Don't type too slowly if people are watching—they'll be able to learn your password. If you mistype, you'll be presented with another chance to login.

Passwords are an important security measure. Don't neglect creating a "good" one. A good password should be easy for you to remember but difficult for others to guess. Words in the dictionary and nicknames are poor choices for a password. One way to generate a password is to use the first letter of each word in a strange yet memorable sentence. For example, *fatIwrnf* could be my password based on the sentence: *For a time I would recommend no forgery.*

If you've typed your login name and password correctly, a short message will appear, called the message of the day. `/etc/motd` This could say anything—the system administrator decides what it should be. After that, a **prompt** appears. It should look something like this:

```
qed:/home/parkinss>
```

The **prompt**, as its name would suggest, is prompting you to enter a command. Every Unix command is a sequence of letters, numbers, and characters. Some valid Unix commands are `ls`, `cat`, and `mv`. Unix is also **case-sensitive**. This means that `cat` and `Cat` are different commands.[2]

The prompt is displayed by a special program called the **shell**. The shell is a special program that acts as a go-between you and the operating system. When you enter a command at the shell prompt the shell interprets your command and calls the program you want. The shell is an extremely useful program that will be the subject of Chapter 4.

By the way, when you first receive your account you will probably be given a temporary password. You should change this to something else. The command to do this is called `passwd`. When you type the command at the prompt, `passwd` prompts you for your old password, and a new password. It also asks you to reenter the new password for validation. Please note that this only effects the computer you are logged into. You may have the same userid on several machines and you will need to repeat this ritual on every computer you have an account on.

2.1.1 A Typical Unix Command

After changing your password the next command to know is `cat`. To use it, type `cat`, and then return:

```
qed:/home/parkinss> cat
```

If you now have a cursor on a line by itself, you've done the correct thing.

[2] In practice, Unix rarely uses the different cases. It is unusual to have a situation where `cat` and `Cat` are different commands.

If you misspelled `cat`, you may have seen

```
qed:/home/parkinss> ct
ct: command not found
qed:/home/parkinss>
```

Thus, the shell informs you that it couldn't find a program named "`ct`" and gives you another prompt to work with. Remember, Unix is case sensitive: `CAT` is a misspelling.

I assume you are now in `cat`. Type anything and hit return. What you should have seen is:

```
qed:/home/parkinss> cat
Shouldn't I be studying economics?
Shouldn't I be studying economics?
```

(The *slanted* text indicates what I typed to `cat`.) What `cat` seems to do is echo the text right back at you. This is useful at times, but isn't right now. So let's get out of this program and move onto commands that have more obvious benefits. Type Ctrl-d to quit `cat`.

To end many Unix commands, type Ctrl-d. Ctrl-d is the end-of-file character, a control character that tells Unix programs that you (or another program) is done entering data. When `cat` sees Ctrl-d it terminates.

For a similar idea, try the program `sort`. As its name indicates, it is a sorting program. If you type a couple of lines, then press Ctrl-d, it will output those lines in a sorted order. In my example I typed the names of some of the buildings at Queen's.

```
qed:/home/parkinss>sort
Stauffer
Mac-Corry
Dunning
```

Then I press Ctrl-d

```
Dunning
Mac-Corry
Stauffer
qed:/home/parkinss>
```

Programs like `cat` and `sort` are called filters because they take in text, filter it, and output the filtered text. `cat` reads in text and performs *no* changes on it. `sort` reads in lines and doesn't output anything until after its seen the EOF character. Many filters run on a line-by-line basis: they will read in a line, perform some computations, and output a different line. When we talk about the shell we will see how filters can make tasks simpler.

2.1.2 Helping Yourself

The `man` command displays reference pages for the command you specify. For example:

```
qed:/home/parkinss> man cat
cat(1)                                                    cat(1)

NAME
  cat - Concatenates or displays files

SYNOPSIS
  cat [-benstuvAET] [--number] [--number-nonblank] [--squeeze-blank]
  [--show-nonprinting] [--show-ends] [--show-tabs] [--show-all]
  [--help] [--version] [file...]

DESCRIPTION
  This manual page documents the GNU version of cat ...
```

There's about one full page of information about `cat`. Try running `man` now. Don't worry if you don't understand the manpage given. Manpages usually assume quite a bit of Unix knowledge—knowledge that I hope you'll have after reading more of this document.

Instead of just letting the text scroll away, `man` stops at the end of each page, waiting for you to decide what to do now. If you just want to go on, press Space and you'll advance a page. If you want to exit (quit) the manual page you are reading, just press q. You'll be back at the shell prompt, and it'll be waiting for you to enter a new command.

There's also a keyword function in `man`. For example, say you're interested in any commands that deal with Postscript, type `man -k ps` or `man -k Postscript`, you'll get a listing of all commands, system calls, and other documented parts of Unix that have the word "ps" (or "Postscript") in their name or short description. This can be very useful when you're looking for a tool to do something, but you don't know its name—or if it even exists.

2.2 Leaving the Computer

If you're done with the computer, but are logged in (you've entered a username and password), first you must logout. To do so, enter the command `exit`. All commands are sent by pressing return. Until you hit return nothing will happen and you can delete what you've done and start over.

If you are sitting in front of a Unix machine, you should **never** turn it off (unless it is on fire). Since Unix is a multi-user system, other people may be logged on or running programs in the background. Unix machines are designed to be run continuosly. Ideally they should only be turned off for hardware or operating system upgrades, and only the system administatrator should turn them off.

If you are accessing qed or another Unix machine from a non-Unix computer you should always log off before turning off the computer you are sitting in front of.

Chapter 3

The Unix Filesystem

In the last chapter, we introduced two filters, `cat` and `sort`. Recall that we typed in some information and the filter printed something on the screen. So that we don't have to type everything in again each time we use a program, Unix provides **files** and **directories** in which to store our information.

Under most operating systems (including Unix), there is the concept of a **file**, which is just a bundle of information given a name (called a **filename**). Examples of files might be your class paper, an e-mail message, or an actual program that can be executed. Essentially, anything saved on disk is saved in an individual file.

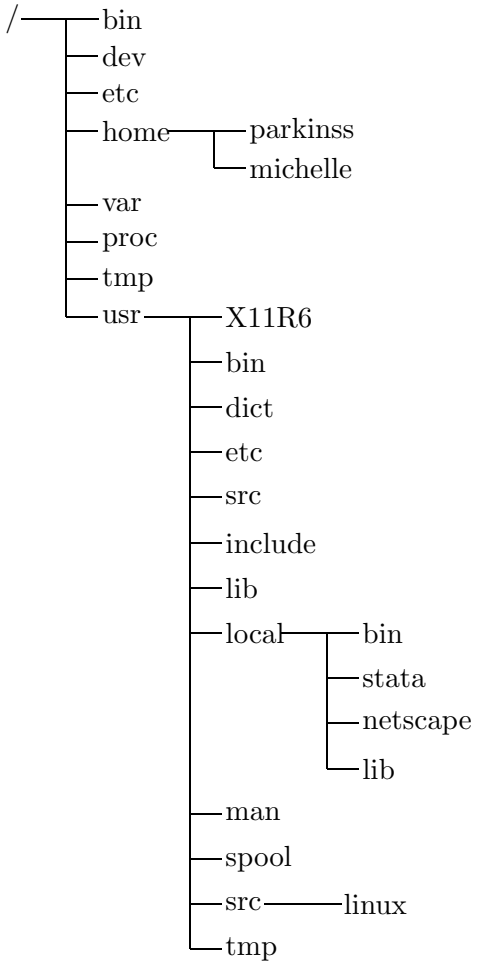
Files are identified by their file names. For example, the file containing the paper you are working on these days might be saved with the file name `working-paper-98-2`. These names usually identify the file and its contents in some form that is meaningful to you. There is no standard format for file names as there is under MS-DOS and some other operating systems; in general, a file name can contain any character (except the `/` character) and is limited to 256 characters in length.

With the concept of files comes the concept of directories. In the Unix filesystem, files are stored under directories. You might think of a directory as a "folder" that contains many different files. In order to identify them, directories are given names. Furthermore, directories are arranged under other directories and so forth, in a tree-like structure.

An example of this type of structure is found by tracing a family's lineage: A couple has a child; that child may have several children; and each of those children may have more children. Of course, we call this structure a family tree. Like the family tree the Unix filesystem is also called a tree, although it is composed of a set of connected files not people. This structure allows users to organize files so they can easily find any particular one.

Each user on our Unix systems starts with one directory. Under this single directory a user can create as many other directories as they like. In each of these 'sub'directories the user can create more directories, and so on—expanding the filesystem to fit their needs. Typically, each subdirectory is dedicated to a single subject and that subject dictates whether a subdirectory should be divided further. Of course, if you prefer, you may organize your files and directories so they are most convenient and useful to you.

Figure 3.1: A Unix directory tree.



Each file and each directory has a name. It has both a short name, which can be the same as another file or directory somewhere else on the system, and a long name which is unique. A short name for a file could be `stata`, while it's "full name" could be `/usr/local/stata`. The full name is usually called the **path**. The path can be decoded into a sequence of directories and it always begins with a backslash. The '\' (on the far left) is called the 'root' directory. The tree 'grows' downward with all paths connected to the root. When you refer to the tree, *up* is towards root and *down* is away from root. For example, here is how `/usr/local/stata` is read:

`/usr/local/stata`
The initial slash indicates the root directory.
Here's the directory `usr`. It's under 'root'.
This is the directory `local`, it's under `usr`.
This is `stata`, it's under `local`.

A path could refer to either a directory or a filename, so `stata` could be either. All the items *before* the short name must be directories. Directories directly connected by a path are called parents (closer to root) and children (farther from root).

To give you a visual representation of the filesystem take a look at Figure 3.1 which shows an abridged directory tree for a Unix filesystem. Like the family tree the filesystem is usually pictured upside down with the root at the top. Notice that you can take any name in the tree and trace a path back to the root. This is called the 'path'. Files are at the end of paths and directories are points in the tree that other paths can branch off from. See the display to the right of the figure for more of the concepts and terms used when referring to and working with the filesystem.

The rest of this chapter introduces you to commands that help you work with files and the filesystem. If you are familiar with MS-DOS, you will already know many commands useful in a hierarchical filesystems, like making directories and so forth. If so, this chapter should be relatively straight forward. Be warned, however, there are many small but important differences between seemingly equivalent Unix and DOS commands. For example, when you delete a file there is no 'undelete' utility in Unix to get it back.

I would recommend that you login to one of the department Unix machines and try the various commands as you read this chapter. In order to make it easier for you to try the commands 'on

the fly', I've included a display of the output that I saw on my screen when I ran the commands on qed. The output you see should be similar. And, before we begin: *Don't be afraid to experiment.* You can't destroy anything by working on the system. Unix has built-in security features to prevent "normal" users from damaging files that are essential to the system. The worst thing that can happen is that you may delete some of your own files. So, at this point, you have nothing to lose.

To give you an idea of where we are going, here is the map for the seven areas we'll cover in the rest of the chapter:

1. Moving about the Filesystem
   - What's in a directory? We've seen that directories contain other directories and files. The command **ls** lists the contents.
   - Where, in the tree, am I? The command **pwd** shows us where we are.
   - How do I navigate? Now that I know where I am how do I move around to other directories? The command **cd** for **c**hange **d**irectory does that.
2. Manipulating the Filesystem
   - The Unix commands **mkdir** and **rmdir** expand and contract the filesystem by **m**a**k**ing and **r**e**m**oving **dir**ectories.
   - Rearranging files. To keep you organized you may copy and move files with **cp** and **mv**.
   - The command **rm** lets you 'clean house' by removing unwanted files.
3. Permissions
   - Remember, Unix is a multiuser operating system. Permissions allow you to protect your files from access by others and to share selected files with certain other users. In this section we'll look at permissions: what they are; how to read them; and how to set and change them.
4. Looking at files.
   - **cat** is a 'quick and dirty' file viewer. Most useful in looking at small files.
   - **more** and **less** are much friendlier file viewers that display files one screenful at a time.
   - **head** and **tail** present only the beginning (head) or end (tail) of the file.
5. Information Commands
   - searching for a word or string in a file is done with the **grep** command.
   - count the number of characters, words or lines in a file with **wc**.
   - The command **diff** compares the contents of two files and shows you where they differ.
6. Miscellaneous things to do with files.
   - **ispell** checks spelling.

- **gzip** compresses files.
- **tar** archives files.
- **mtools** allows you to use the floppy drive on one of the Unix machines so you can use your files at school and home.
- **lpr** sends a file to the printer.

7. A Summary of basic Unix commands.

3.1 Moving about the Filesystem

3.1.1 Looking at Directories with `ls`

The command `ls` lists the files and directory names contained within your current directory. Suppose you are a new user and haven't made any files yet. If you try `ls` as a command, you'll see:

```
qed:/home/parkinss> ls
qed:/home/parkinss>
```

That's right, you'll see nothing. Unix is intensionally terse: it gives you nothing, not even "no files" if there aren't any files. Thus, the lack of output was `ls`'s way of saying it didn't find any files.

But we just saw the directory tree and there were lots of directories: where are they? You've run into the concept of a "current" directory. You can see in your prompt that your current directory is `/home/parkinss`, where you don't have any files. (If you did have files in your current directory `ls` would have displayed their names on the screen). If you want a list of files of a more active directory, try the root directory:

```
qed:/home/parkinss> ls -F /
bin/          etc/          lib/          root/         var/
boot/         floppy/       lost+found/   sbin/         vmlinuz@
cdrom/        home/         mnt/          tmp/          vmlinuz.old@
data@         hwy61/        n/            u@            web/
dev/          initrd/       proc/         usr/
qed:/home/parkinss>
```

In the above command, "`ls -F /`", the directory ("`/`") is a **parameter**. The first word of the command is the command name, and anything after it is a parameter. Parameters generally modify what the program is acting on—for `ls`, the parameters identify which directory you wish to list. Some commands have special parameters called options or switches.

The `-F` is an **option**. An option is a special kind of parameter that starts with a dash and modifies how the program runs, but not what the program runs on. For `ls`, `-F` is an option that lets you see which ones are directories, which ones are special files, which are programs, and which are normal files. We'll talk more about `ls`'s features later.

Now, there are two lessons to be learned here. First, you should learn what `ls` does. Try a few other directories that are shown in Figure 3.1, and see what they contain. Some will be empty, and some will have many files in them. I suggest you try `ls` both with and without the `-F` option. For example, on qed, `ls /usr/local` looks like:

```
qed:/home/parkinss> ls /usr/local

Acrobat3            info          netscape        slatec
bin                 lib           ox              stata
doc                 ls3           rats431         urcdist
i586-unknown-linux  man           sbin            workpap
include             maple         shazam

qed:/home/parkinss>
```

The second lesson is more general. Many Unix commands are like `ls`. They have options, which are generally one character after a dash, and they have parameters. Unlike `ls`, some commands *require* certain parameters and/or options. To show what commands generally look like, we'll use the following form:

`ls` [-alF] [*directory*]

I'll generally use command templates like that before I introduce any command from now on. The first word is the command (in this case `ls`). Following the command are all the parameters. Optional parameters are contained in brackets ("[" and "]"). Meta-variables are *slanted*—they are words that take the place of actual parameters. (For example, above you see *directory*, which should be replaced by the name of a real directory.)

Options are a special case. They are enclosed by brackets, but you can take any one of them without using all of them. For instance, with just the three options given for `ls` you have eight different ways of running the command: with or without each of the options.

3.1.2 The Current Directory and `cd`

`pwd`

Using directories would be cumbersome if you had to type the full path each time you wanted to access a directory. Instead, Unix shells have a feature called the "current" or "present" or "working" directory. In the QED the current directory is displayed in your prompt: `/home/parkinss`. If it doesn't, try the command `pwd`, for **p**resent **w**orking **d**irectory. (In our department the prompt will display the machine name. This is useful in a networked environment with lots of different machines.) `pwd` tells you where you are in the tree.

`cd` [*directory*]

As you can see, `pwd` tells you your current directory—a very simple command. Most commands act, by default, on the current directory. For instance, `ls` without any parameters displays the contents of the current directory. We can change our current directory using `cd`.

At any moment, commands that you enter are assumed to be relative to your **current working directory**. You can think of your working directory as the directory in which you are currently "located". When you first log in, your working directory is set to your home directory—`/home/parkinss`, in our case. Whenever you refer to a file, you may refer to it in relationship to your current working directory, rather than specifying the full pathname of the file.

If you omit the optional parameter *directory*, you're returned to your home, or original, directory. Otherwise, `cd` will change you to the specified directory. For instance suppose the current directory is `/home` then:

```
qed:/home> cd
qed:/home/parkinss> cd /
qed:/> cd home
qed:/home> cd /usr
qed:/usr> cd local/bin
qed:/usr/local/bin>
```

As you can see, `cd` allows you to give either absolute or relative pathnames. An **absolute path** starts with `/` and specifies all the directories before the one you wanted. A **relative path** is in relation to your current directory. In the above example, when I was in `/usr`, I made a relative move to `local/bin`—`local` is a directory under `usr`, and `bin` is a directory under `local`. (`cd home` was also a relative directory change.)

There are two directories used *only* for relative pathnames: "`.`" and "`..`". The directory "`.`" refers to the current directory and "`..`" is the parent directory. These are "shortcut" directories. They exist in *every* directory, but don't really fit the "folder in a folder" concept. The root directory happens to be its own parent. The file `./chapter-1` would be the file called `chapter-1` in the current directory.

The directory "`..`" is most useful in "backing up":

```
qed:/usr/local/bin> cd ..
qed:/usr/local> ls -F ../src
apache-1.1.3/  linux@         redhat/
qed:/usr/local>
```

In this example, I changed to the parent directory using `cd ..`, and I listed the directory `/usr/src` from `/usr/local` using `../src`. Note that if I was in `/home/parkinss`, typing `ls -F ../src` wouldn't do me any good!

The directory `~/` is an alias for your home directory:

```
qed:/usr/local> ls -F ~/
qed:/usr/local>
```

You can see at a glance that there isn't anything in your home directory! `~/` will become more useful as we learn more about how to manipulate files.

3.2 Manipulating the Filesystem

3.2.1 Creating and Removing Directories with `mkdir` and `rmdir`

`mkdir` *directory1* [*directory2 . . . directoryN*]

Creating your own directories is extremely simple under Unix, and can be a useful organizational tool. To create a new directory, use the command `mkdir`. Of course, `mkdir` stands for **m**a**k**e **dir**ectory.

Let's do a small example to see how this works:

```
qed:/home/parkinss> ls -F
qed:/home/parkinss> mkdir Econ853
qed:/home/parkinss> ls -F
Econ853/
qed:/home/parkinss> cd Econ853
qed:/home/parkinss/Econ853>
```

`mkdir` can take more than one parameter, interpreting each parameter as another directory to create. You can specify either the full pathname or a relative pathname; `Econ853` in the above example is a relative pathname.

```
qed:/home/parkinss/Econ853> mkdir /home/parkinss/Econ853/kalman
qed:/home/parkinss/Econ853> mkdir ~/Econ853/graphs
qed:/home/parkinss/Econ853>ls -F
kalman/  graphs/
qed:/home/parkinss/Econ853>
```

`rmdir` *directory1* [*directory2 . . . directoryN*]

The opposite of `mkdir` is `rmdir` (**rm**ove **dir**ectory). `rmdir` works exactly like `mkdir`.

An example of `rmdir` is:

```
qed:/home/parkinss/Econ853>rmdir kalman pics
rmdir: pics: No such file or directory
```

```
qed:/home/parkinss/Econ853>ls -F
graphs/
qed:/home/parkinss/Econ853> cd ..
qed:/home/parkinss> rmdir Econ853
rmdir: Econ853: Directory not empty
qed:/home/parkinss>
```

As you can see, `rmdir` will refuse to remove a non-existent directory, as well as a directory that has anything in it. (Remember, `Econ853` has a subdirectory, `graphs`, in it!) There is one more interesting thing to think about `rmdir`: what happens if you try to remove your current directory? Let's find out:

```
qed:/home/parkinss> cd Econ853
qed:/home/parkinss/Econ853> ls -F
graphs/
qed:/home/parkinss/Econ853> rmdir graphs
qed:/home/parkinss/Econ853> rmdir .
rmdir: .: Operation not permitted
qed:/home/parkinss/Econ853>
```

Another situation you might want to consider is what happens if you try to remove the parent of your current directory. This turns out not to be a problem since the parent of your current directory isn't empty, so it can't be removed!

3.2.2 Copying files with `cp`

`cp` [-i] *source destination*
`cp` [-i] *file1 file2 ... fileN destination-directory*[1]

Be careful with `cp` if you don't have a lot of disk space. No one wants to see a "Disk full" message when working on important files. `cp` can also overwrite existing files without warning—I'll talk more about that danger later.

We'll first talk about the first line in the command template. The first parameter to `cp` is the file to copy—the second is where to copy it. You can copy to either a different filename, or a different directory. Let's try some examples:

```
qed:/home/parkinss> ls -F /etc/passwd
/etc/passwd
qed:/home/parkinss> cp /etc/passwd .
qed:/home/parkinss> ls -F
passwd
```

[1] `cp` has two lines in its template because the meaning of the second parameter can be different depending on the number of parameters.

```
qed:/home/parkinss> cp passwd frog
qed:/home/parkinss> ls -F
frog  passwd
qed:/home/parkinss>
```

The first `cp` command I ran took the file `/etc/passwd`, which contains the names of all the users on the Unix system, and copied it to my home directory. `cp` doesn't delete the source file, so I didn't do anything that could harm the system. So (at least) two copies of `/etc/passwd` exist on my system now, both named `passwd`, but one is in the directory `/etc` and one is in `/home/parkinss`.

Then I created a *third* copy of `/etc/passwd` when I typed `cp passwd frog`—the three copies are now: `/etc/passwd`, `/home/parkinss/passwd` and `/home/parkinss/frog`. The contents of these three files are the same, even if the names aren't.

`cp` can copy files between directories if the first parameter is a file and the second parameter is a directory. In this case, the short name of the file stays the same.

◇ The command `cp` can copy a file and change it's name if both parameters are file names. Here is one danger of `cp`. If I typed `cp /etc/passwd /etc/group`, `cp` would normally create a new file with the contents identical to `passwd` and name it `group`. However, if `/etc/group` already existed, `cp` would destroy the old file without giving you a chance to save it! (The -i option on the `cp` command will ask you to confirm the copy operation if you would destroy a file by copying over it.)

Let's look at another example of `cp`:

```
qed:/home/parkinss> ls -F
frog  passwd
qed:/home/parkinss> mkdir passwd_version
qed:/home/parkinss> cp frog passwd passwd_version
qed:/home/parkinss> ls -F
frog         passwd        passwd_version/
qed:/home/parkinss> ls -F passwd_version
frog  passwd
qed:/home/parkinss>
```

How did I just use `cp`? Evidently, `cp` can take *more* than two parameters. (This is the second line in the command template.) What the above command did is copied all the files listed (`frog` and `passwd`) and placed them in the `passwd_version` directory. In fact, `cp` can take any number of parameters, and interprets the first $n-1$ parameters to be files to copy, and the n^{th} parameter as what directory to copy them too.

You cannot rename files when you copy more than one at a time—they always keep their short name. This leads to an interesting question. What if I type `cp frog passwd toad`, where `frog` and `passwd` exist and `toad` isn't a directory? Try it and see.

3.2.3 Removing Files with `rm`

`rm` [-i] *file1 file2 . . . fileN*

Now that we've learned how to create millions of files with `cp` (and believe me, you'll find new ways to create more files soon), it may be useful to learn how to delete them. Actually, it's very simple, the command you're looking for is `rm`, and it works just like you'd expect. Any file that's a parameter to `rm` gets deleted.

For example:

```
qed:/home/parkinss> ls -F
frog         passwd        passwd_version/
qed:/home/parkinss> rm frog toad passwd
rm: toad: No such file or directory
qed:/home/parkinss> ls -F
passwd_version/
qed:/home/parkinss>
```

◇ As you can see, `rm` is extremely unfriendly. Not only does it not ask you for confirmation, unless the -i option was used, but it will also delete things even if the whole command line wasn't correct. This could actually be dangerous. Consider the difference between these two commands:

```
qed:/home/parkinss> ls -F
toad  frog/
qed:/home/parkinss> ls -F frog
toad
qed:/home/parkinss> rm frog/toad
qed:/home/parkinss>
```

and this

```
qed:/home/parkinss> rm frog toad
rm: frog is a directory
qed:/home/parkinss> ls -F
frog/
qed:/home/parkinss>
```

As you can see, the difference of *one* character made a world of difference in the outcome of the command. It is vital that you check your command lines before hitting return !

3.2.4 Moving files with `mv`

`mv` [-i] *old-name new-name*
`mv` [-i] *file1 file2 . . . fileN new-directory*

Finally, the other file command you should be aware of is `mv`. `mv` looks a lot like `cp`, except that it deletes the original file after copying it. It's a lot like using `cp` and `rm` together. Let's take a look at what we can do:

```
qed:/home/parkinss> cp /etc/passwd .
qed:/home/parkinss> ls -F
passwd
qed:/home/parkinss> mv passwd frog
qed:/home/parkinss> ls -F
frog
qed:/home/parkinss> mkdir report
qed:/home/parkinss> mv frog report
qed:/home/parkinss> ls -F
report/
qed:/home/parkinss> ls -F report
frog
qed:/home/parkinss>
```

As you can see, `mv` will rename a file if the second parameter is a file. If the second parameter is a directory, `mv` will move the file to the new directory, keeping it's shortname the same.

◇ You should be very careful with `mv`. Unless you use the -i option it doesn't check to see if the file already exists, and will remove any old file in its way. For instance, if I had a file named `frog` already in my directory `report`, the command `mv frog report` would delete the file `~/report/frog` and replace it with `~/frog`.

The -i option that I keep refering to makes `rm`, `cp` and `mv` ask you before removing any file. If you use an **alias**, you can make the shell do `rm -i` automatically when you type `rm`. You'll learn more about this later in Section 4.7.3 on page 40.

3.3 Permissions—What Unix remembers *about* a file

In addition to the commands like `cd`, `mv`, and `rm` you learned in Section 3.2, there are other commands that just operate on files but not the data in them. All of these files don't care what is *in* the file—they merely change what Unix remembers about the file.

3.3.1 Concepts of file permissions.

Because there is typically more than one user on a Unix system, Unix provides a mechanism known as **file permissions**, which protect user files from tampering by other users. This mechanism lets files and directories be "owned" by a particular user. For example, because I created the files in my home directory, I own those files and have access to them.

Unix also lets files be shared between users and groups of users. If I desired, I could cut off access to my files so that no other user could access them. However, on most systems the default is to allow other users to read your files but not modify or delete them in any way.

Every file is owned by a particular user. However, files are also owned by a particular **group**, which is a defined group of users of the system. Every user is placed into at least one group when that user's account is created. However, the system administrator may grant the user access to more than one group.

Permissions fall into three main divisions: read, write, and execute. These permissions may be granted to three classes of users: the owner of the file, the group to which the file belongs, and to all users, regardless of group.

Read permission lets a user read the contents of the file, or in the case of directories, list the contents of the directory (using `ls`). Write permission lets the user write to and modify the file. For directories, write permission lets the user create new files or delete files within that directory. Finally, execute permission lets the user run the file as a program or shell script (if the file is a program or shell script). For directories, having execute permission lets the user `cd` into the directory in question.

3.3.2 Interpreting file permissions.

Let's look at an example that demonstrates file permissions. Using the `ls` command with the `-l` option displays a "long" listing of the file, including file permissions.

```
qed:/home/parkinss> ls -l terms

-rw-r--r--   1 parkinss      parkinss           336 Jul 23 17:31 terms

qed:/home/parkinss>
```

The first field in the listing represents the file permissions. The third field is the owner of the file (`parkinss`) and the fourth field is the group to which the file belongs (`users`). Obviously, the last field is the name of the file (`terms`). We'll cover the other fields later.

This file is owned by `parkinss`, and belongs to the group `users`. The string `-rw-r--r--` lists, in order, the permissions granted to the file's owner, the file's group, and everybody else.

The first character of the permissions string ("`-`") represents the type of file. A "`-`" means that this is a regular file. The next three characters ("`rw-`") represent the permissions granted to the file's owner, `parkinss`. The "`r`" stands for "read" and the "`w`" stands for "write". Thus, `parkinss` has read and write permission to the file `terms`.

As mentioned, besides read and write permission, there is also "execute" permission—represented by an "`x`". However, a "`-`" is listed here in place of an "`x`", so parkinss doesn't have execute permission on this file. This is fine, as the file `terms` isn't a program of any kind. Of course, because parkinss owns the file, he may grant himself execute permission for the file if he so desires. (This will be covered shortly.)

The next three characters, ("`r--`"), represent the group's permissions on the file. The group that owns this file is `users`. Because only an "`r`" appears here, any user who belongs to the group `users` may read this file.

The last three characters, also ("`r--`"), represent the permissions granted to every other user on the system (other than the owner of the file and those in the group `users`). Again, because only an "`r`" is present, other users may read the file, but not write to it or execute it.

Here are some other examples of permissions:

`-rwxr-xr-x` The owner of the file may read, write, and execute the file. Users in the file's group, and all other users, may read and execute the file.

`-rw-------` The owner of the file may read and write the file. No other user can access the file.

`-rwxrwxrwx` All users may read, write, and execute the file.

3.3.3 Changing permissions with `chmod`

The command `chmod` is used to set the permissions on a file. Only the owner of a file may change the permissions on that file. The syntax of `chmod` is

---

`chmod` *mode file1 file2 . . . fileN*

Where mode is a combination of the characters [**a,u,g,o**], [**+,-**] and [**r,w,x**]. Briefly, you supply one or more of **a**ll, **u**ser, **g**roup, or **o**ther. Then you specify whether you are adding rights (**+**) or taking them away (**-**). Finally, you specify one or more of **r**ead, **w**rite, and e**x**ecute. Some examples of legal commands are:

`chmod a+r terms`
: Gives all users read access to the file.

`chmod +r terms`
: Same as above—if none of `a`, `u`, `g`, or `o` is specified, `a` is assumed.

`chmod og-x terms`
: Remove execute permission from users other than the owner.

`chmod u+rwx terms`
: Let the owner of the file read, write, and execute the file.

`chmod o-rwx terms`
: Remove read, write, and execute permission from users other than the owner and users in the file's group.

3.4 Looking at files

Two major commands used in Unix for listing files are `cat` and `more`.

3.4.1 Quick file viewing with `cat`

`cat` [-nA] [*file1 file2 ... fileN*]

Try `cat` and you'll see that it's not a user friendly command—it doesn't wait for you to read the file. `cat` is mostly used in conjunction with pipes, something we'll see in chapter 4. However, `cat` does have some useful command-line options. For instance, `n` will number all the lines in the file, and `A` will show control characters as normal characters instead of (possibly) doing strange things to your screen. (Remember, to see some of the stranger and perhaps "less useful" options, use the `man` command: `man cat`.) `cat` will accept input from the keyboard if no files are specified on the command-line.

3.4.2 Nicer file viewing with `more` or with `less`

`more` [-l] [+*linenumber*] [*file1 file2 ... fileN*]

`more` is much more useful than `cat`, and is the command that you might want to use when browsing ASCII text files. `more` will start on a specified linenumber.

Since `more` is an interactive command, I've summarized the major interactive commands below:

Spacebar Moves to the next screen of text.

d This will scroll the screen by 11 lines, or about half a normal, 25-line, screen.

/ Searches for a regular expression. While a regular expression can be quite complicated, you can just type in a text string to search for. For example, `/toad` return would search for the next occurrence of "toad" in your current file. A slash followed by a return will search for the next occurrence of what you last searched for.

: n If you specified more than one file on the command line, this will move to the next file.

q Exits from `more`.

`less` [-l] [+*linenumber*] [*file1 file2 ... fileN*]

The command `less` works just like `more` but it has a few more options, check its `man` page for more.

3.4.3 Looking at the `head` or `tail` of a file

`head` [*-lines*] [*file1 file2 . . . fileN*]

`head` will display the first ten lines in the listed files, or the first ten lines of stdin if no files are specified on the command line. Any numeric option will be taken as the number of lines to print, so `head -15 frog` will print the first fifteen lines of the file `frog`.

`tail` [*-lines*] [*file1 file2 . . . fileN*]

Like `head`, `tail` will display only a fraction of the file. Naturally, `tail` will display the end of the file, or the last ten lines that come through stdin. `tail` also accepts a option specifying the number of lines.

3.5 Information Commands

This section discusses the commands that will alter a file, perform a certain operation on the file, or display statistics on the file.

3.5.1 Searching a text file with `grep`

`grep` [-nvwx] [*-number*] *expression* [*file1 file2 . . . fileN*]

One of the most useful commands in Unix is `grep`. This is a weird name for a utility which searches a text file. The easiest way to use `grep` is like this:

```
qed:/home/parkinss> cat animals
Animals are very interesting creatures. One of my favorite animals is
the tiger, a fearsome beast with large teeth.
I also like the lion---it's really neat!
qed:/home/parkinss> grep iger animals
the tiger, a fearsome beast with large teeth.
qed:/home/parkinss>
```

One disadvantage of this is, although it shows you all the lines containing your word, it doesn't tell you where to look in the file—no line number. Depending on what you're doing, this might be fine. For instance, if you're looking for errors from a program's output, you might try `a.out | grep error`, where `a.out` is your program's name.

If you're interested in where the match(es) are, use the `n` switch to `grep` to tell it to print line numbers. Use the `v` switch if you want to see all the lines that *don't* match the specified expression.

Another feature of `grep` is that it matches only parts of a word, like my example above where `iger` matched `tiger`. To tell `grep` to only match whole words, use the `w`, and the `x` switch will tell grep to only match whole lines.

If you don't specify any files, `grep` will examine stdin.

3.5.2 Counting characters, lines or words with `wc`

`wc` [-clw] [*file1 file2 . . . fileN*]

`wc` stands for **w**ord **c**ount. It simply counts the number of words, lines, and characters in the file(s). If there aren't any files specified on the command line, it operates on stdin.

The three parameters, `clw`, stand for **c**haracter, **l**ine, and **w**ord respectively, and tell `wc` which of the three to count. Thus, `wc -cw` will count the number of characters and words, but not the number of lines. `wc` defaults to counting everything—words, lines, and characters.

One nice use of `wc` is to find how many files are in the present directory: `ls | wc -w`. If you wanted to see how many files that ended with `.tex` there are, try `ls *.tex | wc -w`.

3.5.3 Comparing files with `diff`

`diff` *file1 file2*

In short, `diff` takes two parameters and displays the differences between them on a line-by-line basis. For instance:

```
qed:/home/parkinss> cat frog
Animals are very interesting creatures. One of my favorite animals is
the tiger, a fearsome beast with large teeth.
I also like the lion---it's really neat!
qed:/home/parkinss> cp frog toad
qed:/home/parkinss> diff frog toad
qed:/home/parkinss> cat dog
Animals are very nteresting creatures. One of my favorite animals is

the tiger, a fearsome beast with large teeth.
I also   like the lion---it's really neat!
qed:/home/parkinss> diff frog dog
1c1,2
< Animals are very interesting creatures. One of my favorite animals is
```

```
---
> Animals are very nteresting creatures. One of my favorite animals is
>
3c4
< I also like the lion---it's really neat!
---
> I also   like the lion---it's really neat!
qed:/home/parkinss>
```

As you can see, `diff` outputs nothing when the two files are identical. Then, when I compared two different files, it had a section header, `1c1,2` saying it was comparing line 1 of the left file, `frog`, to lines 1–2 of `dog` and what differences it noticed. Then it compared line 3 of `frog` to line 4 of `dog`. While it may seem strange at first to compare different line numbers, it is much more efficient than listing out every single line if there is an extra return early in one file.

3.6 Miscellaneous commands

3.6.1 Using `ispell` to check your spelling

`ispell` [*file1 file2 ... fileN*]

`ispell` is a very simple Unix spelling program, available in many languages. `ispell` is a filter, like most of the other programs we've talked about, which sucks in an ASCII text file and outputs all the words it considers misspellings. `ispell` operates on the files listed in the command line, or, if there weren't any there, stdin.

`ispell` will offer possible correct spellings and a fancy menu interface if a filename is specified on the command line or will run as a filter-like program if no files are specified.

3.6.2 Archiving with `tar`

The `tar` command is most often used to archive files. Its command syntax is

---

`tar` [xcvtf] [*files*]

For example, the command

```
qed:/home/parkinss>tar cvf backup.tar /etc
```

packs all of the files in `/etc` into the tar archive `backup.tar`. The first argument to `tar`, "`cvf`", is the `tar` "command." `c` tells `tar` to create a new archive file. `v` forces `tar` to use verbose mode, printing each file name as it is archived. The "`f`" option tells `tar` that the next argument, `backup.tar`, is

the name of the archive to create. The rest of the arguments to `tar` are the file and directory names to add to the archive.

The command

```
qed:/home/parkinss>tar xvf backup.tar /etc
```

will extract the tar file `backup.tar` in the current directory.

◇ Old files with the same name are overwritten when extracting files into an existing directory. Before extracting tar files it is important to know where the files should be unpacked because files are extracted with the path name saved in the archive file.

3.6.3 Compressing files with `gzip`

`gzip` [-v#] [*file1 file2 . . . fileN*]
`gunzip` [-v] [*file1 file2 . . . fileN*]

Unlike archiving programs for MS-DOS, `tar` does not automatically compress files as it archives them. If you are archiving two, 1-megabyte files, the resulting tar file is two megabytes in size. The `gzip` command compresses a file (it need not be a tar file).

```
qed:/home/parkinss>gzip -9 backup.tar
```

compresses `backup.tar` and leaves you with `backup.tar.gz`, a compressed version of the file. The `-9` switch tells `gzip` to use the highest compression factor. The `gunzip` command may be used to uncompress a gzipped file.

The commands

`zmore`[*file1 file2 . . . fileN*]
`zgrep` [-nvwx] [-*number*] *expression* [*file1 file2 . . . fileN*]
`zdiff` [*file1 file2*]

---

are used on compressed files. Zmore is a filter which allows examination of compressed or plain text files one screenful at a time. Zgrep is used to invoke the grep on gzip'ed files. Zdiff is used to invoke the diff program on compressed files.

3.6.4 Copying files to and from DOS floppies with `mcopy`

`mcopy` *source destination*

For example,

```
lovell:/home/parkinss>mcopy a:sample.tex .
```

would copy the file `sample.tex` from the DOS floppy I just put in the disk drive on Lovell to my home directory on Lovell.

`mcopy` is part of a package named 'mtools'. Mtools is a collection of tools to allow Unix systems to manipulate MS-DOS files: read, write, and move around files on an MS-DOS filesystem (typically a floppy disk). Where reasonable, each program attempts to emulate the MS-DOS equivalent command.

These tools are likely to be of use to students who use the Linux machines in Dunning 211, have a DOS machine at home and would like to share files between the two. See the man page for mtools for more.

3.7 Printing commands

3.7.1 Printing a file on qed or Frisch

To send a file of ordinary text, a .dvi file, or a Postscript file to be printed on the printer, simply use the lpr command. For example, to send the plain text file myfile.txt in the current directory to the printer, you might issue the following command:

```
qed:\home\parkinss>lpr myfile.txt
```

This command queues the file to be printed on the printer in Dunning 211. You can only print from qed and Frisch. If your file is on another computer use `scp` or `ftp` (See Chapter 9) to copy it to qed and then print from qed.

Checking the Queue

You can check on the status of your print job with the lpq command. The command lpq displays either a message that the queue is empty, or it displays the current job and a list of the jobs waiting behind it to be printed.

Removing Jobs From the Print Queue

Sometimes it's necessary to kill a job you've sent to the printer. This is necessary, for example, if you have sent the wrong file. Or perhaps the printer or the network has malfunctioned and the job is stuck in the queue – you might not be around to retrieve the job once it is printed, and so you wish to remove it for now. In particular, you should always be sure that sensitive materials are removed from the print queue unless you are able to wait by the printer for them to print. If you wish to remove all of your jobs from the queue, enter

```
qed:/home/parkinss>lprm -
```

Printer Courtesy

Please use the printers responsibly. Here are a few tips.

- If you submit a job in error, remove it with lprm as soon as possible.
- Do not send a job to the printer until you put paper in the paper tray.
- Do not assume that if the printer doesn't jump to life as soon as your job is submitted that it is necessary to submit the job again immediately. Some .dvi and PostScript jobs take a long time to print.
- Don't use printers for multiple copies.
- Avoid printing out lots of documentation when it can be read online.

3.8 Summary of basic Unix commands.

Note that options usually begin with "`-`", and in most cases you can specify more than one option with a single "`-`". For example, rather than use the command `ls -l -F`, you can use `ls -lF`.

Rather than listing every option for each command, we only present useful or important commands at this time.

Also note that many of these commands take as arguments a list of files or directories, denoted in this table by "*file1 ... fileN*". For example, the `cp` command takes as arguments a list of files to copy, followed by the destination file or directory. When copying more than one file, the destination must be a directory.

`cd` Change the current working directory.
Syntax: `cd` *directory*
Where *directory* is the directory which you want to change to. ("`.`" refers to the current directory, "`..`" the parent directory. If no directory is specified it defaults to your home directory.)
Example: `cd ../foo` sets the current directory up one level, then back down to `foo`.

`ls` Displays information about the named files and directories.
Syntax: `ls` *files*
Where *files* consists of the the filenames or directories to list. The most commonly used options are `-F` (to display the file type), and `-l` (to give a "long" listing including file size, owner, permissions, and so on).
Example: `ls -lF /home/parkinss` displays the contents of the directory `/home/parkinss`.

`cp` Copies one or more file to another file or directory.
Syntax: `cp` *files destination*
Where *files* lists the files to copy, and *destination* is the destination file or directory.
Example: `cp ../frog joe` copies the file `../frog` to the file or directory `joe`.

`mv` Moves one or more file to another file or directory. This command does the equivalent of a copy followed by the deletion of the original file. You can use this to rename files, like in the MS-DOS command `RENAME`.
Syntax: `mv` *files destination*
Where *files* lists the files to move, and *destination* is the destination file or directory.
Example: `mv ../frog joe` moves the file `../frog` to the file or directory `joe`.

`rm` Deletes files.
Syntax: `rm` *files*
Where *files* describes the filenames to delete.
The `-i` option prompts for confirmation before deleting the file.
Example: `rm -i /home/parkinss/joe /home/parkinss/frog` deletes the files `joe` and `frog` in `/home/parkinss`.

`mkdir` Creates new directories.
Syntax: `mkdir` *dirs*
Where *dirs* are the directories to create.
Example: `mkdir /home/parkinss/test` creates the directory `test` in `/home/parkinss`.

`rmdir` Deletes empty directories. When using `rmdir`, the current working directory must not be within the directory to be deleted.
Syntax: `rmdir` *dirs*
Where *dirs* defines the directories to delete.
Example: `rmdir /home/parkinss/papers` deletes the directory `/home/parkinss/papers`, if empty.

`more` Displays the contents of the named files, one screenful at a time.
Syntax: `more` *files*
Where *files* lists the files to display.
Example: `more papers/history-final` displays the file `papers/history-final`.

`cat` Officially used to concatenate files, `cat` is also used to display the contents of a file on screen.
Syntax: `cat` *files*
Where *files* lists the files to display.
Example: `cat letters` displays the file `letters`.

`grep` Display every line in one or more files that match the given pattern.
Syntax: `grep` *pattern files*
Where *pattern* is a regular expression pattern, and *files* lists the files to search.
Example: `grep Parkinson /etc/passwd` displays the line in that file `/etc/passwd` that contains the pattern `Parkinson`.

Chapter 4

Working with Unix

Unix is a powerful system for those who know how to harness its power. In this chapter, I'll try to describe various ways to use the Unix shell, `bash`, more efficiently.

4.1 Wildcards

In the previous chapter, you learned about the file maintenance commands `cp`, `mv`, and `rm`. Many times you want to deal with more than one file at a time, without typing each filename. For instance, you might want to copy all the files beginning with `data` into a directory called `~/backup`. You could do this by either running many `cp` commands, or you could list every file on one command line. Both of these methods would take a long time, however, and you have a large chance of making an error.

A better way of doing that task is to use a wildcard:

```
qed:/home/parkinss/TSE35> ls
1998-1          1998-3          data1           data3
1998-2          data-new        data2
qed:/home/parkinss/TSE35> mkdir ~/backup
qed:/home/parkinss/TSE35> cp data* ~/backup
qed:/home/parkinss/TSE35> ls -F ~/backup
data-new        data1           data2           data3
qed:/home/parkinss/TSE35>
```

As you can see, the asterisk told `cp` to take all of the files beginning with `data` and copy them to `~/backup`.

Actually, there are a couple of special characters intercepted by the shell, `bash`. The character "`*`", an asterisk, says "replace this word with all the files that will fit this specification". So, the command `cp data*~/backup` gets changed to `cp data-new data1 data2 data3 ~/backup` before it gets run.

To illustrate this, let me reintroduce the command, `echo`. `echo` prints out any parameters. Thus:

```
qed:/home/parkinss> echo Hello!
Hello!
qed:/home/parkinss> echo How are you?
How are you?
qed:/home/parkinss> cd TSE35
qed:/home/parkinss/TSE35> ls
1998-1          1998-3          data1           data3
1998-2          data-new        data2
qed:/home/parkinss/TSE35> echo 199*
1998-1 1998-2 1998-3
qed:/home/parkinss/TSE35> echo *2*
1998-2 data2
qed:/home/parkinss/TSE35>
```

As you can see, the shell expands the wildcard and passes all of the files to the program you tell it to run.

In addition to the asterisk, the shell also interprets a question mark as a special character. A question mark will match one, and only one character. For instance, `ls /etc/??` will display all two letter files in the the `/etc` directory.

4.2 Time Saving with `bash`

4.2.1 Command-Line Editing

Occasionally, you've typed a long command to `bash` and, before you hit return, notice that there was a spelling mistake early in the line. You could just delete all the way back and retype everything you need to, but that takes much too much effort. Instead, you can use the arrow keys to move back there, delete the bad character or two, and type the correct information.

4.2.2 Command and File Completion

Another feature of `bash` is automatic completion of your command lines. For instance, let's look at the following example of a typical `cp` command:

```
qed:/home/parkinss> ls -F
this-is-a-long-file
qed:/home/parkinss> cp this-is-a-long-file short
qed:/home/parkinss> ls -F
short              this-is-a-long-file
qed:/home/parkinss>
```

It's a big pain to have to type every letter of `this-is-a-long-file` whenever you try to access it. Instead of typing the whole filename, type `cp th` and press and release the Tab. Like magic, the rest of the filename shows up on the command line, and you can type in `short`.

When you hit Tab, `bash` looks at what you've typed and looks for a file that starts like that. For instance, if I type `/usr/bin/ema` and then hit Tab, `bash` will find `/usr/bin/emacs` since that's the only file that begins `/usr/bin/ema` on my system. However, if I type `/usr/bin/ld` and hit Tab, `bash` beeps at me. That's because three files, `/usr/bin/ld`, `/usr/bin/ldd`, and `/usr/bin/ld86` all start with `/usr/bin/ld` on my system.

If you try a completion and `bash` beeps, you can immediately hit Tab again to get a list of all the files your start matches so far. That way, if you aren't sure of the exact spelling of your file, you can start it and scan a much smaller list of files.

4.3 The Standard Input and The Standard Output

Let's try to tackle a simple problem: getting a listing of the `/usr/bin` directory. If all we do is `ls /usr/bin`, some of the files scroll off the top of the screen. How can we see all of the files?

4.3.1 Unix Concepts

The Unix operating system makes it very easy for programs to use the terminal. When a program writes something to your screen, it is using something called **standard output**. Standard output, abbreviated as stdout, is how the program writes things to a user. The name for what you tell a program is **standard input** (stdin). It's possible for a program to communicate with the user without using standard input or output, but most of the commands I cover in this book use stdin and stdout.

For example, the `ls` command prints the list of the directories to standard output, which is normally "connected" to your terminal. An interactive command, such as your shell, `bash`, reads your commands from standard input.

In this section, we're going to examine three ways of fiddling with the standard input and output: input redirection, output redirection, and pipes.

4.3.2 Output Redirection

A very important feature of Unix is the ability to **redirect** output. This allows you, instead of viewing the results of a command, to save it in a file or send it directly to a printer. For instance, to redirect the output of the command `ls /usr/bin`, we place a `>` sign at the end of the line, and say what file we want the output to be put in:

```
qed:/home/parkinss> ls
qed:/home/parkinss> ls -F /usr/bin > listing
qed:/home/parkinss> ls
listing
qed:/home/parkinss>
```

As you can see, instead of writing the names of all the files, the command created a totally new file in your home directory. Let's try to take a look at this file using the command `cat`. If you think back, you'll remember `cat` was a fairly useless command that copied what you typed (the standard input) to the terminal (the standard output). `cat` can also print a file to the standard output if you list the file as a parameter to `cat`:

```
qed:/home/parkinss> cat listing
...
qed:/home/parkinss>
```

The exact output of the command `ls /usr/bin` appeared in the contents of `listing`. All well and good, although it didn't solve the original problem.

However, `cat` does do some interesting things when it's output is redirected. What does the command `cat listing > newfile` do? Normally, the `> newfile` says "take all the output of the command and put it in `newfile`." The output of the command `cat listing` is the file `listing`. So we've invented a new (and not so efficient) method of copying files.

How about the command `cat > fox`? `cat` by itself reads in each line typed at the terminal (standard input) and prints it right back out (standard output) until it reads Ctrl-d. In this case, standard output has been redirected into the file `fox`. Now `cat` is serving as a rudimentary editor:

```
qed:/home/parkinss> cat > todo
Pay PUC invoice and return library books.
press Ctrl-d
qed:/home/parkinss>
```

We've now created the file `todo` that contains the sentence "Pay PUC invoice and return library books." One last use of the versatile `cat` command is to con**cat**enate files together. `cat` will print out every file it was given as a parameter, one after another. So the command `cat listing todo` will print out the directory listing of `/usr/bin`, and then it will print out my errands for the day. Thus, the command `cat listing todo> listandtodo` will create a new file containing the contents of both `listing` and `todo`.

4.3.3 Input Redirection

Like redirecting standard output, it is also possible to redirect standard input. Instead of a program reading from your keyboard, it will read from a file. Since input redirection is related to output redirection, it seems natural to make the special character for input redirection be <, it, too, is used after the command you wish to run.

This is generally useful if you have a data file and a command that expects input from standard input. Most commands also let you specify a file to operate on, so < isn't used as much in day-to-day operations as other techniques.

4.3.4 The Pipe

Many Unix commands produce a large amount of information. For instance, it is not uncommon for a command like `ls /usr/bin` to produce more output than you can see on your screen. In order for you to be able to see all of the information that a command like `ls /usr/bin` prints, it's necessary to use another Unix command, one like `more`.[1]

However, that doesn't help the problem that `ls /usr/bin` displays more information than you can see. `more < ls /usr/bin` won't work—input redirection only works with files, not commands! You *could* do this:

```
qed:/home/parkinss> ls /usr/bin > temp-ls
qed:/home/parkinss> more temp-ls
...
qed:/home/parkinss> rm temp-ls
```

However, Unix supplies a much cleaner way of doing that. You can just use the command `ls /usr/bin | more`. The character "`|`" indicates a **pipe**. Like a water pipe, a Unix pipe controls flow. Instead of water, we're controlling the flow of information!

A useful tool with pipes are programs called **filters**. A filter is a program that reads the standard input, changes it in some way, and outputs to standard output. `more` is a filter—it reads the data that it gets from standard input and displays it to standard output one screen at a time, letting you read the file. `more` isn't a great filter because its output isn't suitable for sending to another program.

The commands `cat`, `sort`, `head`, and `tail` are also filters. For instance, if you wanted to read only the first ten lines of the output from `ls`, you could use `ls /usr/bin | head`.

4.4 Multitasking

4.4.1 Using Job Control

Job control refers to the ability to put processes (another word for programs, essentially) in the **background** and bring them to the **foreground** again. That is to say, you want to be able to make something run while you go and do other things, but have it be there again when you want to tell it something or stop it. In Unix, the main tool for job control is the shell—it will keep track of jobs for you, if you learn how to speak its language.

The two most important words in that language are `fg`, for foreground, and `bg`, for background. To find out how they work, use the command `yes` at a prompt.

```
qed:/home/parkinss> yes
```

[1] `more` is named because that's the prompt it originally displayed: `--more--`. In many versions of Unix the `more` command is similar to a different, but more advanced command (called `less`).

This will have the startling effect of running a long column of `y`'s down the left hand side of your screen, faster than you can follow. To get them to stop, you'd normally type ctrl-c to kill it, but instead you should type ctrl-z this time. It appears to have stopped, but there will be a message before your prompt, looking more or less like this:

```
[1]+   Stopped                 yes
```

It means that the process `yes` has been suspended in the background. You can get it running again by typing `fg` at the prompt, which will put it into the foreground again. If you wish, you can do other things first, while it's suspended. Try a few `ls`'s or something before you put it back in the foreground.

Once it's returned to the foreground, the `y`'s will start coming again, as fast as before. You do not need to worry that while you had it suspended it was "storing up" more `y`'s to send to the screen: when a program is suspended the whole program doesn't run until you bring it back to life. (Now type ctrl-c to kill it for good, once you've seen enough).

Let's pick apart that message we got from the shell:

```
[1]+   Stopped                 yes
```

The number in brackets is the **job number** of this job, and will be used when we need to refer to it specifically. (Naturally, since job control is all about running multiple processes, we need some way to tell one from another). The `+` following it tells us that this is the "current job" — that is, the one most recently moved from the foreground to the background. If you were to type `fg`, you would put the job with the `+` in the foreground again. (More on that later, when we discuss running multiple jobs at once). The word `Stopped` means that the job is "stopped". The job isn't dead, but it isn't running right now. Unix has saved it in a special suspended state, ready to jump back into the action should anyone request it. Finally, the `yes` is the name of the process that has been stopped.

Before we go on, let's kill this job and start it again in a different way. The command is named `kill` and can be used in the following way:

```
qed:/home/parkinss> kill %1
[1]+  Terminated               yes
qed:/home/parkinss>
```

There you have it—the job has been terminated.

Now, start `yes` running again, like this:

```
qed:/home/parkinss> yes > /dev/null
```

If you read the section about input and output redirection, you know that this is sending the output of `yes` into the special file `/dev/null`. `/dev/null` is a black hole that eats any output sent to it.

After typing this, you will not get your prompt back, but you will not see that column of `y`'s either. Although output is being sent into `/dev/null`, the job is still running in the foreground. As usual, you can suspend it by hitting ctrl-z. Do that now to get the prompt back.

```
qed:/home/parkinss> yes > /dev/null
["yes" is running, and we just typed ctrl-z]
[1]+  Stopped                 yes >/dev/null

qed:/home/parkinss>
```

Hmm...is there any way to get it to actually *run* in the background, while still leaving us the prompt for interactive work? The command to do that is `bg`:

```
qed:/home/parkinss> bg
[1]+ yes >/dev/null  &
qed:/home/parkinss>
```

Now, you'll have to trust me on this one: after you typed `bg`, `yes > /dev/null` began to run again, but this time in the background. You can do anything you want at the prompt, and `yes` will happily continue to sending its output into the black hole.

There are now two different ways you can kill it: with the `kill` command you just learned, or by putting the job in the foreground again and hitting it with an interrupt, ctrl-c. Let's try the second way, just to understand the relationship between `fg` and `bg` a little better;

```
qed:/home/parkinss> fg
yes >/dev/null

[now it's in the foreground again.  Imagine that I hit ctrl-c to terminate it]

qed:/home/parkinss>
```

There, it's gone. Now, start up a few jobs running in simultaneously, like this:

```
qed:/home/parkinss> yes  > /dev/null &
[1] 1024
qed:/home/parkinss> yes | sort > /dev/null &
[2] 1026
qed:/home/parkinss> yes | uniq > /dev/null
[and here, type ctrl-z to suspend it, please]

[3]+  Stopped                 yes | uniq >/dev/null
qed:/home/parkinss>
```

The first thing you might notice about those commands is the trailing `&` at the end of the first two. Putting an `&` after a command tells the shell to start in running in the background right from the very beginning. (It's just a way to avoid having to start the program, type ctrl-z, and then

type `bg`.) So, we started those two commands running in the background. The third is suspended and inactive at the moment. You may notice that the machine has become slower now, as the two running ones require some amount of CPU time.

Each one told you it's job number. The first two also showed you their process identification numbers, or PID's, immediately following the job number. The PID's are normally not something you need to know, but occasionally come in handy.

Let's kill the second one, since I think it's making your machine slow. You could just type `kill %2`, but that would be too easy. Instead, do this:

```
qed:/home/parkinss> fg %2
yes | sort >/dev/null
[type ctrl-c to kill it]

qed:/home/parkinss>
```

As this demonstrates, `fg` takes parameters beginning with `%` as well. In fact, you could just have typed this:

```
qed:/home/parkinss> %2
yes | sort >/dev/null
[type ctrl-c to kill it]

qed:/home/parkinss>
```

This works because the shell automatically interprets a job number as a request to put that job in the foreground. It can tell job numbers from other numbers by the preceding `%`. Now type `jobs` to see which jobs are left running:

```
qed:/home/parkinss> jobs
[1]-  Running                 yes >/dev/null  &
[3]+  Stopped                 yes | uniq >/dev/null
qed:/home/parkinss>
```

The "`-`" means that job number 1 is second in line to be put in the foreground, if you just type `fg` without giving it any parameters. The "`+`" means the specified job is first in line—a `fg` without parameters will bring job number 3 to the foreground. However, you can get to it by naming it, if you wish:

```
qed:/home/parkinss> fg %1
yes >/dev/null
[now type ctrl-z to suspend it]

[1]+  Stopped                 yes >/dev/null
qed:/home/parkinss>
```

Having changed to job number 1 and then suspending it has also changed the priorities of all your jobs. You can see this with the `jobs` command:

```
qed:/home/parkinss> jobs
[1]+  Stopped                 yes >/dev/null
[3]-  Stopped                 yes | uniq >/dev/null
qed:/home/parkinss>
```

Now they are both stopped (because both were suspended with ctrl-z), and number 1 is next in line to come to the foreground by default. This is because you put it in the foreground manually, and then suspended it. The "+" always refers to the most recent job that was suspended from the foreground. You can start it running again:

```
qed:/home/parkinss> bg
[1]+ yes >/dev/null  &
qed:/home/parkinss> jobs
[1]-  Running                 yes >/dev/null
[3]+  Stopped                 yes | uniq >/dev/null
qed:/home/parkinss>
```

Notice that now it is running, and the other job has moved back up in line and has the `+`. Now let's kill them all so your system isn't permanently slowed by processes doing nothing.

```
qed:/home/parkinss> kill %1 %3
[1]+  Terminated              yes >/dev/null
[3]   Terminated              yes | uniq >/dev/null
qed:/home/parkinss>
```

You should see various messages about termination of jobs—nothing dies quietly, it seems. Figure 4.1 shows a quick summary of what you should know for job control.

4.4.2 The Theory of Job Control

It is important to understand that job control is done by the shell. There is no program on the system called `fg`; rather, `fg`, `bg`, `&`, `jobs`, and `kill` are all shell-builtins. This is a logical way to do it: since each user wants their own job control space, and each user already has their own shell, it is easiest to just have the shell keep track of the user's jobs. Therefore, each user's job numbers are meaningful only to that user: my job number [1] and your job number [1] are probably two totally different processes. In fact, if you are logged in more than once, each of your shells will have unique job control data, so you as a user might have two different jobs with the same number running in two different shells.

The way to tell for sure is to use the Process ID numbers (`PID`'s). These are system-wide — each process has its own unique `PID` number. Two different users can refer to a process by its `PID` and know that they are talking about the same process (assuming that they are logged into the same machine!)

`fg` *%job* — This is a shell command that returns a job to the foreground. To find out which one this is by default, type `jobs` and look for the one with the `+`.
Parameters: Optional job number. The default is the process identified with `+`.

`&` — When an `&` is added to the end of the command line, it tells the command to run in the background automatically. This process is then subject to all the usual methods of job control detailed here.

`bg` *%job* — This is a shell command that causes a suspended job to run in the background. To find out which one this is by default, type `jobs` and look for the one with the `+`.
Parameters: Optional job number. The default is the process identified with `+`.

`kill` *%job PID* — This is a shell command that causes a background job, either suspended or running, to terminate. You should always specify the job number or PID, and if you are using job numbers, remember to precede them with a `%`.
Parameters: Either the job number (preceded by `%`) or PID (no `%` is necessary). More than one process or job can be specified on one line.

`jobs` — This shell command just lists information about the jobs currently running or suspending. Sometimes it also tells you about ones that have just exited or been terminated.

ctrl-c — This is the generic interrupt character. Usually, if you type it while a program is running in the foreground, it will kill the program (sometimes it takes a few tries). However, not all programs will respond to this method of termination.

ctrl-z — This key combination usually causes a program to suspend, although a few programs ignore it. Once suspended, the job can be run in the background or killed.

Figure 4.1: A summary of commands and keys used in job control.

Let's take a look at one more command to understand what `PID`s are. The `ps` command will list all running processes, including your shell. Try it out. It also has a few options, the most important of which (to many people) are `a`, `u`, and `x`. The `a` option will list processes belonging to any user, not just your own. The `x` switch will list processes that don't have a terminal associated with them.[2] Finally, the `u` switch will give additional information about the process that is frequently useful.

To really get an idea of what your system is doing, put them all together: `ps -aux`. You can then see the process that uses the more memory by looking at the `%MEM` column, and the most CPU by looking at the `%CPU` column. (The `TIME` column lists the *total* amount of CPU time used.)

Another quick note about PIDs. `kill`, in addition to taking options of the form *%job#*, will take options of raw PIDs. So, put a `yes > /dev/null` in the background, run `ps`, and look for `yes`. Then type `kill PID`.[3]

[2] This only makes sense for certain system programs that don't have to talk to users through a keyboard.
[3] In general, it's easier to just kill the job number instead of using PIDs.

4.5 Process priority with using nice

If your run a command that makes large demands on the computer, you should consider running it at a lower priority. On a multi-user system, it is considerate of other users to run "nice" jobs, rather than computer intensive jobs that slow down the system for other users. The `nice` command runs a process at a lower priority. The format is:

`nice` *command*

---

where *command* is the name of the command you want to issue.

4.6 Using nohup to keep jobs running after you logout

The `nohup` command allows you to leave jobs running after you logout.

Here is an example:

```
qed:/home/parkinss> nohup shazam < myjob.in > myjob.out &
```

would run the SHAZAM batch file `myjob.in`, put the output in the file `myjob.out` and continue to run after you logout.

Of course you might also want to run your job at lower system priority with the nice command:

```
qed:/home/parkinss> nohup nice shazam < myjob.in > myjob.out &
```

4.7 bash Customization

One of the cool things about Unix is that the system's designers did not attempt to predict every need that we might have; instead, they tried to make it easy for us to tailor the environment to their own particular needs. This is mainly done through **configuration files**. These are also known as "init files", "rc files" (for "run control"), or "dot files", because the filenames often begin with ".".

The most important configuration files are the ones used by the shell. Linux's default shell is `bash`, and that's the shell this chapter covers. Before we go into how to customize `bash`, we should know what files `bash` looks at.

4.7.1 Shell Startup

There are several different ways `bash` can run. It can run as a **login shell**, which is how it runs when you first login. The login shell should be the first shell you see.

Another way `bash` can run is as an **interactive shell**. This is any shell which presents a prompt to a human and waits for input. A login shell is also an interactive shell. A way you can get a

non-login interactive shell is, say, a shell inside `xterm`. Any shell that was created by some other way besides logging in is a non-login shell.

Finally, there are **non-interactive shells**. These shells are used for executing a file of commands, much like MS-DOS's batch files—the files that end in `.BAT`. These **shell scripts** function like mini-programs.

Depending on the type of shell, different files will be used at shell startup:

Type of Shell	Action
Interactive login	The file `.bash_profile` is read and executed
Interactive	The file `.bashrc` is read and executed
Non-interactive	The shell script is read and executed

4.7.2 Startup Files

If you would like to have the same environment no matter what type of interactive shell you start put the command "`source ~/.bashrc`" into your `.bash_profile`: The `source` command tells the shell to interpret the argument as a shell script. What it means for us is that every time `.bash_profile` is run, `.bashrc` is *also* run.

Now, we'll just add commands to our `.bashrc`. If you ever want a command to only be run when you login, add it to your `.bash_profile`.

4.7.3 Aliasing

What are some of the things you might want to customize? In the file /etc/profile on qed you will find the line

```
alias dir="ls -l"
```

That command defined a shell **alias** called `dir` that "expands" to the normal shell command "`ls -l`" when invoked by the user. So, you can just type `dir` to get the effect of "`ls -l`" in only three quarters the keystrokes. What happens is that when you type `dir` and hit return, `bash` intercepts it, because it's watching for aliases, replaces it with "ls -l", and runs that instead. There is no actual program called `dir` on the system, but the shell automatically translated the alias into a valid program.

You may want to look at the file /etc/profile on qed (use the `more` command) to see what aliases you already have. If you need something else you may define your own.

Some sample aliases are in Figure 4.7.3. You could put them in your own `.bashrc`. One especially interesting alias is the first one. With that alias, whenever someone types `ls`, they automatically have a `-F` flag tacked on. (The alias doesn't try to expand itself again.) This is a common way of adding options that you use every time you call a program.

```
alias ls="ls -F"          # give characters at the end of listing
alias dir="ls -l"          # special ls
alias la="ls -a"
alias ro="rm *~; rm .*~"  # this removes backup files created by Emacs
alias fraser="ssh -l jsp fraser.sfu.ca"
```

Figure 4.2: Some sample aliases for `bash`.

Variable name	Contains	Example
`HOME`	Your home directory	`/home/parkinss`
`TERM`	Your terminal type	`xterm`, `vt100`, or `console`
`SHELL`	The path to your shell	`/bin/bash`
`USER`	Your login name	`parkinss`
`PATH`	A list to search for programs	`/bin:/usr/bin:/usr/local/bin:/usr/bin/X11`

Figure 4.3: Some important environment variables.

Notice the comments with the `#` character in Figure 4.7.3. Whenever a `#` appears, the shell ignores the rest of the line.

Knowing how to make your own aliases is probably at least half of all the shell customization you'll ever do. Experiment a little, find out what long commands you find yourself typing frequently, and make aliases for them. You'll find that it makes working at a shell prompt a much more pleasant experience.

Another major thing one does in a `.bashrc` is set **environment variables**. Every program executes in an **environment**, and that environment is defined by the shell that called the program. The environment could be said to exist "within" the shell. Changing your environment involves the `export` command :

You may think of the `export` command as meaning "Please export this variable out to the environment where I will be calling programs, so that its value is visible to them."

The command `env` will list all the environment variables. It's possible, especially if you're using X, that the list will scroll off the screen. If that happens, just pipe `env` through `more`: `env | more`.

A few of these variables can be fairly useful, so I'll cover them. Look at Figure 4.7.3. Those four variables are defined automatically when you login: you don't set them in your `.bashrc` or `.bash_login`.

Another variable, `PATH`, is also crucial to the proper functioning of the shell. Here's mine:

```
qed:/home/parkinss> env | grep ^PATH
PATH=/bin:/usr/bin:/usr/local/bin:/usr/bin/X11
qed:/home/parkinss>
```

Your `PATH` is a colon-separated list of the directories the shell should search for programs, when you type the name of a program to run. When I type `ls` and hit return, for example, the `bash` first

looks in `/bin` and there it has a hit! `/bin/ls` does exist and is executable, so `bash` stops searching for a program named `ls` and runs it. There might well have been another `ls` sitting in the directory `/usr/bin`, but `bash` would never run it unless I asked for it by specifying an explicit pathname:

```
qed:/home/parkinss> /usr/bin/ls
```

The `PATH` variable exists so that we don't have to type in complete pathnames for every command. When you type a command, Bash looks for it in the directories named in `PATH`, in order, and runs it if it finds it.

There's a lot more to configuring your `.bashrc`, and not enough room to explain it here. You can read the `bash` man page for more, or look at a bash manual.

Chapter 5

The X Window System

This chapter is geared to people who intend to use the X Window System. If you are in Dunning 211, sitting at `Frisch`, `Lovell`, `Waugh`, `Cox` or `Wald` – or using `laffer` in the grad lounge then you may use X.

5.1 What is The X Window System?

For lack of a good description, think of X as something that allows you to use a graphical interface on Unix. X will be easier to describe if we learn a few terms first. The **client** is an X program. For instance, `xterm` is probably the client that displays your shell when you log on. The **server** is a program that provides services to the client program. For instance, the server draws the window for `xterm` and communicates with the user.

Since the client and the server are two separate programs, it is possible to run the client and the server *on two physically separate machines*. In addition to supplying a standard method of doing graphics, you can run a program on a remote machine (across the country, if you like) and have it display on the workstation right in front of you.

A third term you should be familiar with is the **window manager**. The window manager is a special client that tells the server where to position various windows and provides a way for the user to move these windows around. The server, by itself, does nothing for the user. It is merely there to provide a buffer between the user and the client.

So X is a server that controls the window manager client that in turn provides the look and feel of your graphical interface.

5.2 Starting X

If X doesn't start automatically when you login, it is possible to start it from the regular text-mode shell prompt. The command to start X is `startx`.

5.3 What's This on my Screen?

When you first start X, several programs are started. First, the server is started. Then, several clients are usually started. Unfortunately, this is not standardized across different machines. It is likely that among these clients are a window manager, `fvwm`, a prompt, `xterm`, and maybe a clock, `xclock` and some other X applications.

5.3.1 X applications

An X application is one that uses X. There are many programs that take advantage of X. Some programs, like `Emacs`, can be run either as a text-mode program *or* as a program that creates its own X window. However, most X programs can only be run under X. Some X applications other than Emacs that you are likely to use include `Netscape` for browsing; `gv` for viewing Postscript and PDF files; `xdvi` to view dvi files; `xmaple` if you want to use maple; and so on, there are many X applications. A special one called `xterm` is the subject of the next subsection.

5.3.2 XTerm

The window with a prompt in it is being controlled by a program called `xterm`. `xterm` emulates a terminal so that regular text-mode Unix applications work correctly.

For much of this book, we're going to be learning about the Unix command-line, and you'll find that inside your `xterm` window. In order to type into `xterm`, you *usually* have to move your mouse cursor (possibly shaped like an "X" or an arrow) into the `xterm` window.

One way of starting more programs under X is through an `xterm`. Since X programs are standard Unix programs, they can be run from normal command prompts such as `xterm`s. Since running a long program from a `xterm` would tie up the `xterm` as long as the program was running, people normally start X programs in the background. For more information about this, see Section 4.4.

5.3.3 Window Managers

A *window manager* is the program that controls the look and feel of the graphical interface. It defines the appearance of the windows and how you operate them. Here are some things the window manager controls:

Focus

The first thing you'll be interested in is **focus**. The focus of the server is which window will get what you type into the keyboard. Usually in X the focus is determined by the position of the mouse cursor. If the mouse cursor is in one `xterm`'s window[1], that `xterm` will get your keypresses. This is different from many other windowing systems, such as Microsoft Windows, OS/2, or the Macintosh,

[1] You can have more then one copy of `xterm` running at the same time.

where you must click the mouse in a window before that window gets focus. Usually under X, if your mouse cursor wanders from a window, focus will be lost and you'll no longer be able to type there.

Of course, it is possible to configure the window manager so that you must click on or in a window to gain focus, and click somewhere else to lose it, identical to the behavior of Microsoft Windows.

Moving Windows

Another very configurable thing in X is how to move windows around. On the machines in the Economics Department the easiest way to move windows is to move the mouse cursor onto the **title bar** click on the left mouse button and drag the window around the screen.

Depth

Since windows are allowed to overlap in X, there is a concept of **depth**. Even though the windows and the screen are both two dimensional, one window can be in front of another, partially or completely obscuring the rear window. **Raising** the window, or bringing it to the front, is usually accomplished by clicking on a window's title bar with any of the the mouse buttons. **Lowering** the window, or pushing it to the back, is done in the same way. Clicking on the title bar of a raised window lowers it, and vice-versa.

Menus

Another purpose for window managers is for them to provide menus for the user to quickly accomplish tasks that are done over and over. For instance, I might make a menu choice that automatically launches `Netscape` or an additional `xterm` for me. That way I don't need to type the program name in an `xterm`. To view a menu position the mouse cursor outside of a window (on the desktop) and click one of the mouse buttons.

5.4 Things common to all X programs

5.4.1 Geometry

There are a few things common to all programs running under X. In X, the concept of **geometry** is where and how large a window is. A window's geometry has four components:

- The horizontal size, usually measured in pixels although some applications, like `xterm` and `emacs`, measure their size in terms of number of characters they can fit in the window.
- The vertical size, also usually measured in pixels. It's possible for it to be measured in characters.

- The horizontal distance from one of the sides of the screen. For instance, `+35` would mean make the left edge of the window thirty-five pixels from the left edge of the screen. On the other hand, `-50` would mean make the right edge of the window fifty pixels from the right edge of the screen. It's generally impossible to start the window off the screen, although a window can be moved off the screen. (The main exception is when the window is very large.)
- The vertical distance from either the top or the bottom. A positive vertical distance is measured from the top of the screen; a negative vertical distance is measured from the bottom of the screen.

All four components get put together into a geometry string that looks like: `503x73-78+0`. (That translates into a window 503 pixels long, 73 pixels high, put near the top right hand corner of the screen.) Another way of stating it is *hsize*`x`*vsize*±*hplace*±*vplace*.

5.4.2 Display

X Windows allows clients running on remote computers to display information on your screen. This is done through two mechanisms. The first is the DISPLAY environment variable, which tells the client where on the network your X Server is running. The second is referred to as access control. Access control lets you decide which remote computers are allowed to send information to your screen (i.e., open up windows), through your X Server. We'll try a few examples using the public machine *frisch*.

If you are logged into an X Terminal, or have started X Windows on your workstation, you can determine what the DISPLAY environment variable is set to by typing:

```
frisch:/home/parkinss>echo $DISPLAY
:0.0
frisch:/home/parkinss>
```

In this case, the DISPLAY environment variable points to the X Terminal named :0.0. The part to the left of the colon tells us where the X Server is running with respect to the network (on frisch in this case). The part to the right of the colon (0.0) tells which monitor at this address to display the results on. In general, the 0.0 part will never change. If you need a different setting, you are far too advanced for this discussion of X Windows!

To determine who is allowed access to your display, that is, which computers can display information through your X Server, you use the xhost command. To see which computers can display information on your screen, type:

```
frisch:/home/parkinss>xhost
access control enabled, only authorized clients can connect
INET:waugh.econ.queensu.ca
INET:sargan.econ.queensu.ca
INET:edith.econ.queensu.ca
INET:qed.econ.queensu.ca
frisch:/home/parkinss>
```

The first line of output says that the access control system is in effect, and only clients running on machines you specify may display information on the screen. The second through fifth lines specify the names of some computers which are allowed to connect to the display (i.e., open windows on the display).

Even though the machine frisch is not mentioned in the machine list, it can still open the display since it is the computer which I logged in to.

To illustrate the use of DISPLAY and xhost, the next example will show a user on frisch log into the computer named lovell and open an xterm running on lovell. See Chapter 9 for details on logging in to a remote machine.

```
frisch:/home/parkinss> echo $DISPLAY
:0.0
frisch:/home/parkinss>telnet lovell
Trying 130.15.74.4...
Connected to lovell.econ.queensu.ca.
Escape character is '^]'.
Password:
Linux lovell 2.0.34 #1 Fri Sep 4 16:00:42 EDT 1998 i586 unknown
Last login: Tue Sep  8 10:42:45 on ttyp1 from frisch.econ.QueensU.CA.
No mail.
lovell:/home/parkinss> xterm
Xlib: connection to "frisch:0.0" refused by server
Xlib: Client is not authorized to connect to Server
xterm Xt error: Can't open display: frisch:0.0
lovell:/home/parkinss>
```

This error means that lovell does not know which X Server to send the display information to. This can be corrected by setting the DISPLAY environment variable.

```
lovell:/home/parkinss>export DISPLAY=frisch:0.0
lovell:/home/parkinss> xterm
Xlib: connection to "frisch:0.0" refused by server
Xlib: Client is not authorized to connect to Server
xterm Xt error: Can't open display: frisch:0.0
```

The second error means that lovell is not allowed to display information through the X Server pointed to by DISPLAY. This can be corrected by going back to frisch, adding lovell to the list of machines which can connect to the X Server, and returning to lovell's prompt.

```
frisch:/home/parkinss>xhost + lovell.econ.queensu.ca
lovell being added to access control list
frisch:/home/parkinss>
```

Once lovell is added as a host, telnet to lovell, set the display and away you go.

The use of the DISPLAY environment variable and the xhost command illustrates the network transparency of the X Window system. With these mechanisms, any computer on the Internet can display applications on your local machine, as if you were logged in directly to the remote computer. This type of access also has its drawbacks. Any user on the Internet can set their DISPLAY environment variable to point to your display. This means anybody on the Internet can display anything on your screen, unless you take the proper precautions and set the access control list. The following is a list of common uses of the xhost command. Usually, you should use the most restricted form of access control that you can.

To allow X clients from a specific compute server access to your display use:

```
frisch:/home/parkinss> xhost + hostname.domain
```

Almost every X client takes a special command line argument called -display. Using this argument you can set what display a particular X client will use. Of course the machine running the client must already be added to the access control list using xhost. For example:

```
frisch:/home/parkinss>xterm -display lovell.econ.queensu.ca:0.0
```

will cause an xterm from frisch to display on the lovell X Terminal.

Every X application has a display that it is associated with. The display is the name of the screen that the X server controls. A display consists of three components:

- The machine name that the server is running on. At stand-alone Linux installations the server is always running on the same system as the clients. In such cases, the machine name can be omitted.
- The number of the server running on that machine. Since any one machine could have multiple X servers running on it (unlikely for most Linux machines, but possible) each must have a unique number.
- The screen number. X supports a particular server controlling more than one screen at a time. You can imagine that someone wants a lot of screen space, so they have two monitors sitting next to each other. Since they don't want two X servers running on one machine for performance reasons, they let one X server control both screens.

These three things are put together like so: *machine*:*server-number*.*screen-number*
(eg qed.econ.queensu.ca:0.0).

Users of ssh (see Chapter 9) will find that using X applications across machines is a snap and no DISPLAY variable need be set.

5.5 Copying and Pasting text

Copying and pasting text between X applications is very easy to do with a mouse. First you must select the text that you wish to copy. Place the mouse cursor in front of the first character you wish

Name	Followed by	Example
`-geometry`	geometry of the window	`xterm -geometry 80x24+0+90`
`-display`	display you want the program to appear	`xterm -display Chimera:0.0`
`-fg`	the primary foreground color	`xterm -fg yellow`
`-bg`	the primary background color	`xterm -bg blue`

Figure 5.1: Standard options for X programs.

to copy, press and hold the left mouse button, and drag the mouse cursor over the text you want to copy. You can drag over several lines if you wish. Release the left mouse button when all the text you wish to copy is highlighted. (if you single click the mouse before holding you drag the cursor over single characters, if you double click you drag over words, triple click you drag over lines—just like on a macintosh)

To paste the text move the mouse cursor to location the text is to be pasted (this could be in the same window different location or a different window altogether) then click the middle mouse button.

5.6 Exiting X

Depending on how X is configured, there are two possible ways you might have to exit X. The first is if your window manager controls whether or not X is running. If it does, you'll have to exit X using a menu (see Section 5.3.3 on page 45). To display a menu, click a button on the background. The important menu entry should be "Exit Window Manager" or "Exit X" or some entry containing the word "Exit".

The other method would be for a special `xterm` to control X. If this is the case, there is probably a window labeled "login". To exit from X, move the mouse cursor into that window and type "exit".

Chapter 6

File Editors

In order to get anything done on a computer, you need a way to put text into files, and a way to change text that's already in files. An **editor** is a program for doing this. If you just want to look at a file, but not change it, a utility like `less` (see chapter 3) is more suitable than an editor.

Pico is a simple, easy-to-use text editor with a layout very similar to the Pine mailer (see Chapter 8). Pico is a particularly good choice for an editor is you do not require the flexibility of the Emacs editor, or if you wish to make a minor change to a file and don't want to wait for Emacs to start up. Since Pico and Pine work in the same way, the remainder of this chapter will focus on the Emacs editor, see Chapter 8 for more on how to use pico.

6.1 What's Emacs?

`Emacs` is one of the most popular editors around—partly because it's very easy for a complete beginner to get actual work done with it.

To learn `Emacs`, you need to find a file of plain text (letters, numbers, and the like), copy it to your home directory[1] and invoke `Emacs` on the file:

```
qed:/home/parkinss> emacs new.qed
```

"Invoking" Emacs can have different effects depending on where where you do it. From a plain console displaying only text characters, Emacs will just take over the whole console. If you invoke it from X, Emacs will actually bring up its own window (X is introduced in chapter 5). I will assume that you are doing it from a text console, but everything carries over logically into the X Windows version—just substitute the word "window" in the places I've written "screen". Also, if you are using X remember that you have to move the mouse pointer into Emacs's window to type in it.

Your screen (or window, if you're using X) should now resemble Figure 6.1. Most of the screen contains your text document, but the last two lines are especially interesting if you're trying to learn Emacs. The second-to-last line (the one with the long string of dashes) is called the **mode line**.

[1] For instance, if you were working on qed `cp /usr/local/doc/new.qed .`

Figure 6.1: Emacs was just started with `emacs /usr/local/doc/new.qed`

In my mode line, you see "Top". It might be "All" instead, and there may be other minor differences. The line immediately below the mode line is called the **minibuffer**. Emacs uses the minibuffer to flash messages at you, and occasionally uses it to read input from you, when necessary. Ignore it for now; we won't be making much use of the minibuffer for a while.

Before you actually change any of the text in the file, you need to learn how to move around. The cursor should be at the beginning of the file, in the upper-left corner of the screen. To move forward, type `C-f` (that is, hold down the Control key while you press "f", for "forward"). It will move you forward a character at a time, and if you hold both keys down, your system's automatic key-repeat should take effect in a half-second or so. `C-b` (for "backward") has the opposite behavior. And, while we're at it, `C-n` and `C-p` take you to the next and previous lines, respectively.[2]

Using the control keys is usually the quickest way of moving around when you're editing. The goal of `Emacs` is to keep your hands over the alpha-numeric keys of the keyboard, where most of your work gets done. However, if you want to, the page-up, page-down and arrow keys should also work. When you're using X, you are able to position the mouse pointer and click with the left button to move the cursor where you want.

[2]In case you hadn't noticed yet, many of Emacs's movement commands consist of combining Control with a single mnemonic letter.

Use `C-p` and `C-b` to get all the way back to the upper-left corner. Now keep `C-b` held a little longer. You should hear an annoying bell sound, and see the message "`Beginning of buffer`" appear in the minibuffer. At this point you might wonder, "But what is a buffer?"

When Emacs works on a file, it doesn't actually work on the file itself. Instead, it copies the contents of the file into a special Emacs work area called a **buffer**, where you can modify it to your heart's content. When you are done working, you tell Emacs to save the buffer—in other words, to write the buffer's contents into the corresponding file. Until you do this, the file remains unchanged, and the buffer's contents exist only inside of Emacs.

Until now, everything we have done has been "non-destructive." As you type in the buffer take a look at the beginning of the mode line at the bottom of the screen. When you change the buffer so that its contents are no longer the same as those of the file on disk, Emacs displays two asterisks at the beginning of the mode line, to let you know that the buffer has been modified:

```
--**-Emacs: some_file.txt          (Fundamental)--Top-----------------------
```

These two asterisks are displayed as soon as you modify the buffer, and remain visible until you save the buffer. You can save the buffer multiple times during an editing session—the command to do so is just `C-x C-s` (hold down Control and hit "x" and "s" while it's down. It's deliberately easy to type, because saving your buffers is something best done early and often.

I'm going to list a few more commands now, along with the ones you've learned already, and you can practice them however you like. I'd suggest becoming familiar with them before going any further:

`C-f`	Move forward one character.
`C-b`	Move backward one character.
`C-n`	Go to next line.
`C-p`	Go to previous line.
`C-a`	Go to beginning of line.
`C-e`	Go to end of line.
`C-v`	Go to next page/screenful of text.
`C-l`	Redraw the screen, with current line in center.
`C-d`	Delete this character (practice this one).
`C-k`	Delete text from here to end of line.
`C-x C-s`	Save the buffer in its corresponding file.
Backspace	Delete preceding character (the one you just typed).

6.2 Getting Started Quickly in X

If all you're interesting in is editing a few files quickly, an X user doesn't have to go much further beyond the menus at the top of the screen: (These menus are not available in text mode.)

The Buffers menu lists the different files you've been editing in this incarnation of Emacs. The File menu shows a bunch of commands for loading and saving files—many of them will be described later. The Edit menu displays some commands for editing one buffer, and the Help menu provides on-line documentation.

You'll notice keyboard equivalents are listed next to the choices in the menu. Since, in the long run, they'll be quicker, you might want to learn them. Also, for better or for worse, most of Emacs's functionality is *only* available through the keyboard—you might want to read the rest of this chapter.

6.3 Editing Many Files at Once

Emacs can work on more than one file at a time. In fact, the only limit on how many buffers your Emacs can contain is the actual amount of memory available on the machine. The command to bring a new file into an Emacs buffer is `C-x C-f`. When you type it, you will be prompted for a filename in the minibuffer:

```
Find file: ~/
```

The syntax here is the same one used to specify files from the shell prompt; slashes represent subdirectories, ~means your home directory. You also get **filename completion**, meaning that if you've typed enough of a filename at the prompt to identify the file uniquely, you can just hit Tab to complete it (or to show possible completions, if there are more than one). Once you have the full filename in the minibuffer, hit return, and Emacs will bring up a buffer displaying that file. In Emacs, this process is known as **finding** a file. Go ahead and find some other unimportant text file now, and bring it into Emacs (do this from our original buffer `new.qed`). Now you have a new buffer; I'll pretend it's called `another_file.txt`, since I can't see your mode line.

Your original buffer seems to have disappeared—you're probably wondering where it went. It's still inside Emacs, and you can switch back to it with `C-x b`. When you type this, you will see that the minibuffer prompts you for a buffer to switch to, and it names a default. The default is the buffer you'd get if you just hit return at the prompt, without typing a buffer name. The default buffer to switch to is always the one most recently left, so that when you are doing a lot of work between two buffers, `C-x b` always defaults to the "other" buffer (which saves you from having to type the buffer name). Even if the default buffer is the one you want, however, you should try typing in its name anyway.

Notice that you get the same sort of completion you got when finding a file: hitting Tab completes as much of a buffer name as it can, and so on. Whenever you are being prompted for something in the minibuffer, it's a good idea to see if Emacs is doing completion. Taking advantage of completion whenever it's offered will save you a lot of typing. Emacs usually does completion when you are choosing one item out of some predefined list.

Everything you learned about moving around and editing text in the first buffer applies to the new one. Go ahead and change some text in the new buffer, but don't save it (i.e. don't type

`C-x C-s`). Let's assume that you want to discard your changes without saving them in the file. The command for that is `C-x k`, which "kills" the buffer. Type it now. First you will be asked which buffer to kill, but the default is the current buffer, and that's almost always the one you want to kill, so just hit return. Then you will be asked if you *really* want to kill the buffer—Emacs always checks before killing a buffer that has unsaved changes in it. Just type "yes" and hit return, if you want to kill it.

Go ahead and practice loading in files, modifying them, saving them, and killing their buffers. Try to have at least five buffers open at once, so you can get the hang of switching between them.

6.4 Ending an Editing Session

When you are done with your work in Emacs, make sure that all buffers are saved that should be saved, and exit Emacs with `C-x C-c`. Sometimes `C-x C-c` will ask you a question or two in the minibuffer before it lets you leave—don't be alarmed, just answer them in the obvious ways.

6.5 The Meta Key

You've already learned about one "modifier key" in Emacs, the Control key. There is a second one, called the **Meta** key, which is used almost as frequently. Your keyboard's Alt key should be the Meta key.

The notation `M-x` is analogous to `C-x` (substitute any character for "`x`").

6.6 Cutting, Pasting, Killing and Yanking

Emacs, like any good editor, allows you to cut and paste blocks of text. In order to do this, you need a way to define the start and end of the block. In Emacs, you do this by setting two locations in the buffer, known as **mark** and **point**. To set the mark, go to the place you want your block to begin and type `C-SPC` ("`SPC`" means Space, of course). You should see the message "Mark set" appear in the minibuffer. The mark has now been set at that place. There will be no special highlighting indicating that fact, but you know where you put it, and that's all that matters.

What about **point**? Well, it turns out that you've been setting point every time you move the cursor, because "point" just refers to your current location in the buffer. In formal terms, point is the spot where text would be inserted if you were to type something. By setting the mark, and then moving to the end of the block of text, you have actually defined a block of text. This block is known as the **region**. The region always means the area between mark and point.

Merely defining the region does not make it available for pasting. You have to tell Emacs to copy it in order to be able to paste it. To copy the region, make sure that mark and point are set correctly, and type `M-w`. It has now been recorded by Emacs. In order to paste it somewhere else, just go there and type `C-y`. This is known as **yanking** the text into the buffer.

If you want to actually move the text of the region to somewhere else, type `C-w` instead of `M-w`. This will **kill** the region—all the text inside it will disappear. In fact, it has been saved in the same way as if you had used `M-w`. You can yank it back out with `C-y`, as always. The place Emacs saves all this text is known as the **kill-ring**. Some editors call it the "clipboard" or the "paste buffer".

There's another way to do cutting and pasting: whenever you use `C-k` to kill to the end of a line, the killed text is saved in the kill-ring. If you kill more than one line in a row, they are all saved in the kill-ring together, so that the next yank will paste in all the lines at once. Because of this feature, it is often faster to use repeated `C-k`'s to kill some text than it is to explicitly set mark and point and use `C-w`. However, either way will work. It's really a matter of personal preference how you do it.

6.7 Searching and Replacing

There are several ways to search for text in Emacs. Many of them are rather complex, and not worth going into here. The easiest and most entertaining way is to use **isearch**. "Isearch" stands for "incremental search". Suppose you want to search for the string "gadfly" in the following buffer:

```
I was growing afraid that we would run out of gasoline, when my passenger
exclaimed ``Gadzooks!  There's a gadfly in here!''.
```

You would move to the beginning of the buffer, or at least to some point that you know is before the first occurrence of the goal word, "gadfly", and type `C-s`. That puts you in isearch mode. Now start typing the word you are searching for, "gadfly". But as soon as you type the "g", you see that Emacs has jumped you to the first occurrence of "g" in the buffer. If the above quote is the entire contents of the buffer, then that would be the first "g" of the word "growing". Now type the "a" of "gadfly", and Emacs leaps over to "gasoline", which contains the first occurrence of a "ga". The "d" gets you to gadzooks, and finally, "f" gets you to "gadfly", without your having had to type the entire word.

What you are doing in an isearch is defining a string to search for. Each time you add a character to the end of the string, the number of matches is reduced, until eventually you have entered enough to define the string uniquely. Once you have found the match you are looking for, you can exit the search with return or any of the normal movement commands. If you think the string you're looking for is behind you in the buffer, then you should use `C-r`, which does an isearch backwards.

If you encounter a match, but it's not the one you were looking for, then hit `C-s` again while still in the search. This will move you forward to the next complete match, each time you hit it. If there is no next match, it will say that the search failed, but if you press `C-s` again at that point, the search will wrap around from the beginning of the buffer. The reverse holds true for `C-r` — it wraps around the end of the buffer.

Try bringing up a buffer of plain English text and doing and isearch for the string "`the`". First you'd type in as much as you wanted, then use repeated `C-s`'s to go to all instances of it. Notice that it will match words like "`them`" as well, since that also contains the substring "`the`". To search

only for "`the`", you'd have to do add a space to the end of your search string. You can add new characters to the string at any point in the search, even after you've hit `C-s` repeatedly to find the next matches. You can also use Backspace or Delete to remove characters from the search string at any point in the search, and hitting return exits the search, leaving you at the last match.

Emacs also allows you to replace all instances of a string with some new string—this is known as **query-replace**. To invoke it, type `query-replace` and hit return. Completion is done on the command name, so once you have typed "query-re", you can just hit Tab to finish it. Say you wish to replace all instances of "gadfly" with "housefly". At the "`Query replace:` " prompt, type "gadfly", and hit return. Then you will be prompted again, and you should enter "housefly". Emacs will then step through the buffer, stopping at every instance of the word "gadfly", and asking if you want to replace it. Just hit "`y`" or "`n`" at each instance, for "Yes" or "No", until it finishes. If this doesn't make sense as you read it, then try it out.

What's Really Going On Here?

Actually, all these **keybindings** you have been learning are shortcuts to Emacs functions. For example, `C-p` is a short way of telling Emacs to execute the internal function `previous-line`. However, all these internal functions can be called by name, using `M-x`. If you forgot that `previous-line` is bound to `C-p`, you could just type `M-x previous-line` return, and it would move you up one line. Try this now, to understand how `M-x previous-line` and `C-p` are really the same thing.

The designer of Emacs started from the ground up, first defining a whole lot of internal functions, and then giving keybindings to the most commonly-used ones. Sometimes it's easier just to call a function explicitly with `M-x` than to remember what key it's bound to. The function `query-replace`, for example, is bound to `M-%` in some versions of Emacs. But who can remember such an odd keybinding? Unless you use query-replace extremely often, it's easier just to call it with `M-x`.

6.8 Asking Emacs for Help

Emacs has extensive help facilities—so extensive, in fact, that we can only touch on them here. The most basic help features are accessed by typing `C-h` and then a single letter. For example, `C-h k` gets help on a key (it prompts you to type a key, then tells you what that key does). `C-h t` brings up a short Emacs tutorial. Most importantly, `C-h C-h C-h` gets you help on help, to tell you what's available once you have typed `C-h` the first time. If you know the name of an Emacs function (`save-buffer`, for example), but can't remember what key sequence invokes it, then use `C-h w`, for "`where-is`", and type in the name of the function. Or, if you want to know what a function does in detail, use `C-h f`, which prompts for a function name.

Another source of information is the **Info** documentation reader. Info is too complex a subject to go into here, but if you are interested in exploring it on your own, type `C-h i` and read the paragraph at the top of the screen. It will tell you how get more help.

6.9 Specializing Buffers: Modes

Emacs buffers have **modes** associated with them. The reason for this is that your needs when writing a mail message are very different from your needs when, say, writing a program. Rather than try to come up with an editor that would meet every single need all the time (which would be impossible), the designer of Emacs chose to have Emacs behave differently depending on what you are doing in each individual buffer. Thus, buffers have modes, each one designed for some specific activity. The main features that distinguish one mode from another are the keybindings, but there can be other differences as well.

The most basic mode is `fundamental` mode, which doesn't really have any special commands at all. The programming modes are more interesting.

6.10 Programming Modes

6.10.1 C Mode

If you use Emacs for programming in the C language, you can get it to do all the indentation for you automatically. Files whose names end in "`.c`" or "`.h`" are automatically brought up in `c-mode`. This means that certain special editing commands, useful for writing C-programs, are available. In C-mode, Tab is bound to `c-indent-command`. This means that hitting the Tab key does not actually insert a tab character. Instead, if you hit Tab anywhere on a line, Emacs automatically indents that line correctly for its location in the program. This implies that Emacs knows something about C syntax, which it does (although nothing about semantics—it cannot insure that your program has no errors!)

In order to do this, it assumes that the previous lines are indented correctly. That means that if the preceding line is missing a parenthesis, semicolon, curly brace, or whatever, Emacs will indent the current line in a funny way. When you see it do that, you will know to look for a punctuation mistake on the line above.

You can use this feature to check that you have punctuated your programs correctly—instead of reading through the entire program looking for problems, just start indenting lines from the top down with Tab, and when something indents oddly, check the lines just before it. In other words, let Emacs do the work for you!

6.10.2 TEXMode

Emacs has a special TEX mode for editing input files. It provides facilities for check the balance of delimiters and for TEXing a file. We'll talk a bit about AUCTEX an extension to TEX Mode.

AUCTEX is a comprehensive environment for writing input files for LATEX using Emacs. AUCTEX lets you run TEX/LATEX and other LATEX-related tools, such as ispell, xdvi, a syntax checker, etcfrom inside Emacs. This is much more convenient than entering commands at the prompt.

Especially 'running LaTeX' is interesting. When the command (C-c C-c) is executed the Emacs view is split in two, and the output of LaTeX is printed in the second half of the screen, as you may simultaneously continue editing your document. In case TeX found any errors when processing your input you can call the function TeX-next-error (C-c ') which will move the cursor to the first given error, and display a short explanatory text along with the message TeX gave. This procedure may be repeated until all errors have been displayed. Once you've successfully formatted your document, you may preview or print it.

When it comes to editing AUCTeX automatically indents your 'LaTeX-source', not only as you write it—you can also let it indent and format an entire document.

A number of more or less intelligent keyboard macros have been defined to aid you editing your document. In Chapter 7 we saw that changing to boldface involved the command `\textbf{}` AUCTeX has this function bound to the keystrokes (C-c C-f C-b). Pressing this sequence will give you the bold command and place the cursor between the braces ready for you to type.

Apart from these special features, AUC TeX provides an large range of handy Emacs macros, which in several different ways can help you write your LaTeX documents fast and painless. All features of AUCTeX are documented using the Emacs online documentation system. Look there for more.

6.11 Being Even More Efficient

When you're moving around, you'll want to use the fastest means available. You know that `C-f` is `forward-char`, `M-f` is `forward-word`, `C-b` is `backward-char`, guess what `M-b` does? That's not all, though: you can move forward a sentence at a time with `M-e`, as long as you write your sentences so that there are always two spaces following the final period (otherwise Emacs can't tell where one sentence ends and the next one begins). `M-a` is `backward-sentence`.

`C-e` gets you to the end of the line and use `C-a` to go to the beginning of the line. Use `C-v` to screen down and `M-v` to screen up.

When you have to type in a filename, no need to type in the whole name. Just type in enough of it to identify it uniquely, and let Emacs's completion finish the job by hitting Tab or Space.

If you are typing some kind of plain text, and somehow your auto-filling (or auto-wrapping) has gotten screwed up, use `M-q`, which is `fill-paragraph` in common text modes. This will "adjust" the paragraph you're in as if it had been wrapped line by line, but without your having to go mess around with it by hand. `M-q` will work from inside the paragraph, or from its very beginning or end.

Chapter 7

LaTeX

TeX is well known to be *the* typesetting package, and a vast cult of TeX lovers has evolved. But to the beginning TeX user, or to someone wondering if they should bother changing to TeX it is often not clear what all the fuss is about. After all, are not both WordPerfect and Word capable of high quality output? Newcomers have often already seen what TeX is capable of (many books, journals, letters are now prepared with TeX) and so expect to find a tremendously powerful and friendly package. In fact they do, but that fact is well hidden in one's initial TeX experiences. In this chapter we describe LaTeX, which is a version of TeX that features some built in macros for chapter and section breaking, auto-generation of a table of contents, indexing, tables, figures, cross referencing, itemized and enumerated lists, and more—think of LaTeX as TeX with a few built in conveniences. As we continue I hope to show you a little of what makes LaTeX great, and why other packages cannot even begin to compete. Be warned that a little patience is required—LaTeX's virtues are rather subtle to begin with. But when the penny drops, you will wonder how you ever put up with anything different.

LaTeX is a typesetter, not a word-processor. LaTeX was designed with no limiting application in mind. It was intended to be able to prepare practically any document—from a single page all-text letter to a full blown book with huge numbers of formulae, tables, figures etc. The size and the complexity of a LaTeXable document is limited only by hardware considerations. Furthermore, LaTeX seeks to achieve all this while setting typesetting standards of the highest order for itself. The expertise of generations of professional printers has been captured in LaTeX, and it has been taught all the tricks of the trade.

Historically, printers prepared a document by placing metal characters in a large tray and arranging and binding them to form a page. This was very precisely done, but the ultimate precision was limited because of the mechanical nature of things and by time considerations. LaTeX prepares a page in an analogous manner (putting your characters and formulae into "boxes" which are then "glued" together to form the page), but has the advantage of enormous precision because placement calculations are performed by computer.

"But conventional word processors run on computers , too", you object. Yes, but their fundamental limitation is that they try to "keep up" with you and "typeset" your document as you type.

This means that it can only make decisions at a local level (eg, it decides where to break a line just as you type the end of the line). LaTeX's secret is that it waits until you have typed the whole document before it typesets a single thing! This means that LaTeX can make decisions of a global nature in order to optimize the aesthetic appeal of your document. It has been taught what looks good and what looks bad (having been given a measure of the "badness" of various possibilities) and makes choices for your document that are designed to make it "minimally bad".

But LaTeX's virtues run much deeper than that, which is just as well because it is possible to get satisfactory, though imperfect, results from some word processors. One of LaTeX's strongest points is its ability to typeset complicated formulae with ease. Not only does LaTeX make hundreds of special symbols easily accessible, it will lay them out for you in your formulae. It has been taught all the spacing, size and font conventions that printers have decided look best in typeset formulae. Although, of course, it doesn't understand any mathematics it knows the grammar of mathematics—it recognizes binary relations, binary operators, unary operators, etc. and has been taught how these parts should be set. It is consequently rather difficult to get an equation to look bad in LaTeX.

Another advantage of compiling a document after it is typed is that cross-referencing can be done. You can label and refer back to chapters, sections, tables etc. by name rather than absolute number, and LaTeX will number and cross-reference these for you. Similarly, it will compile a table of contents, glossary, index and bibliography for you.

Essential to the spirit of LaTeX is that it formats the document while you just take care of the content, making for increased productivity. The cross-referencing just mentioned is just part of this. Many more labor-saving mechanisms are provided for through style files. These are generic descriptions of classes of documents, teaching LaTeX just how each class likes to be formatted. This is taught in terms of font preferences, default page sizes, placement of title, author, date, etc. For instance, a paper style file could teach LaTeX that when typesetting a theorem it should embolden the part that states the theorem number and typeset the text of the theorem statement in slanted Roman typeface (as in many journals). The typist simply provides an indication that a theorem is being stated, and then types the text of the theorem without bothering to choose any fonts or do any formatting—all that is done by the style file. Style files exist for all manner of document—letters, articles, papers, books, proceedings, review articles, and so on.

In addition to style files, there are macro packages. A macro is just a definition of a new LaTeX command in terms of existing ones. Don't think small when you think of macros! When typing a document that has a lot of repetition in it, say the same expression is used again and again in different equations, you can define a macro in your document to abbreviate that expression. But macros can teach LaTeX how to typeset all sorts of complicated structures, not just parts of an equation. Many macro packages (files that are just collections of definitions) have been written to teach LaTeX all sorts of applications. LaTeX itself is a macro package.

Another facet of the design of LaTeX allows it to use practically any output device. In fact, LaTeX doesn't talk to any printers, screens or phototypesetters at all! Instead, when a document is compiled a device independent (.dvi) is produced— LaTeX does not compile with any particular output device in mind. Printer drivers are then invoked on this .dvi file and, in consultation with

the font data for that printer, produce output suitable for the particular device. You can choose an HP Laserjet driver, or an Apple LaserWriter driver, or a dot matrix driver etc. All use the same .dvi file as input (and remember the material in there is set to enormous accuracy) and attempt to image that file on the particular device as faithfully as possible. If you are using a top of the line laser printer or phototypesetter, then TEX's massive internal precision will not be wasted. Alternatively, a dot matrix printer will give a coarse approximation of the ideal image that is suitable only for proof-reading. In addition to portability, these .dvi files help ensure that there are very few printing surprises when you move from one device to another: how many times has your favorite word-processor made you reformat a document when you wish to change printers?

The novice reader will still have no idea of what a LATEX source file looks like. Indeed, why do we keep referring to it as a source file? The fact of the matter is that LATEX is essentially a programming language. Just as in any compiled language (e.g., Pascal, C) one prepares a source file and submits it to the compiler which attempts to produce an object file (.dvi file in the TEX case). To learn LATEX is to learn the command syntax of the commands that can be used in the source file.

LATEX was designed to run on a multitude of computers. It is therefore the case that the documentation for LATEX is not computer specific. Only command syntax is described—i.e., the content of your source file—but few details of how to get from there to a printout are given. Remember, because the input file is plain text you can prepare the input file on any computer with an editor regardless of which operating system is run.

The quality of the document is not affected because of the careful design of LATEX—whether you work on a a machine with massive floating point precision or a modest 286 the .dvi files produced on compilation will be identical; and when those files are submitted to printer equivalent printer drivers the output will be identical because the font information they draw on is identical. By the nature of TEX most time is spent editing the source document (before submitting it for compilation). No special interface is necessary here, you just use your favorite text editor (Emacs is a particularly good one). Whatever the editor, there are just a few basic steps to preparing a document:

1. Choose a document class to base your document on (e.g., letter, article).

2. Glance through the material you have to type, and decide what definitions might be made to save you a lot of time. Also, decide on the overall structure of the prospective document (e.g., will the largest sectional unit be a chapter or a part?). If you are going to compose as you type, then pause a moment to think ahead and plan the structure of your document. The importance of this step cannot be overstressed, for it makes clear in your mind what you want from TEX.

3. Prepare your input file, specifying only the content and the logical structure (parts, sections, theorems,...) thereof and forgetting about formatting details.

4. Submit your input or source file to the TEX compiler for compilation of a .dvi file.

5. If the compiler finds anything in your source file strongly objectionable, say incorrect command syntax, then return to editing.

6. Run a previewer to preview your compiled document on the screen (like xdvi, for example). Resolution is only limited by your screen, and can be very good indeed on some modern monitors.

7. Go back to editing your document until glaring errors have been taken care of.

8. Make a printout of your compiled document, and check for those errors that you failed to notice on the screen.

Performing these steps may be effected through typing at the system prompt (bare-bones technique) or through choosing menu options in Emacs.

Before discussing the eight steps any further look at the example LaTeX file in Figure 7.1. This is a slightly modified copy of the standard LaTeX example file `SMALL.TEX`. The line numbers down the left-hand side are not part of the file, but have been added to make it easier to identify various portions. Also have a look at Figure 7.2 which shows, more or less, the result of processing this file.

7.1 Step 1. Choose a document class

The first information LaTeX needs to know when processing an input file is the type of document the author wants to create. The four standard document classes available in LaTeX are in Table 7.1. These document classes can be modified by a number of *style options* the standard options are listed in Table 7.2.

Table 7.1: Document Classes

`article`	for articles in scientific journals, presentations, short reports, program documentation, invitations ...
`report`	for longer reports containing several chapters, small books, ...
`book`	for real books
`slides`	for slides. This class uses a large sans serif font.

As an example, an input file for a LaTeX document could start with the following line:

```
\documentclass[11pt]{article}
```

this instructs LaTeX to typeset the document as an *article* with a font size of *eleven points*.

```
 1: % SMALL.TEX -- Released 5 July 1985, Modified Aug 4 1998 for LaTeX2e
 2: % USE THIS FILE AS A MODEL FOR MAKING YOUR OWN LaTeX INPUT FILE.
 3: % EVERYTHING TO THE RIGHT OF A  %  IS A REMARK TO YOU AND IS IGNORED
 4: % BY LaTeX.
 5: %
 6: % WARNING!  DO NOT TYPE ANY OF THE FOLLOWING 10 CHARACTERS EXCEPT AS
 7: % DIRECTED:        &   $   #   %   _   {   }   ^   ~   \
 8:
 9: \documentclass[11pt]{article}  % YOUR INPUT FILE MUST CONTAIN THESE
10: \begin{document}                  % TWO LINES PLUS THE \end COMMAND AT
11:                                   % THE END
12:
13: \section{Simple Text}          % THIS COMMAND MAKES A SECTION TITLE.
14:
15: Words are separated by one or    more      spaces.  Paragraphs are
16:     separated by one or more blank lines.  The output is not affected
17: by adding extra spaces or extra blank lines to the input file.
18:
19:
20: Double quotes are typed like this: ``quoted text''.
21: Single quotes are typed like this: `single-quoted text'.
22:
23: Long dashes are typed as three dash characters---like this.
24:
25: Italic text is typed like this: \textit{this is italic text}.
26: Bold   text is typed like this: \textbf{this is  bold  text}.
27:
28: \subsection{A Warning or Two}        % THIS MAKES A SUBSECTION TITLE.
29:
30: If you get too much space after a mid-sentence period---abbreviations
31: like etc.\ are the common culprits)---then type a backslash followed by
32: a space after the period, as in this sentence.
33:
34: Remember, don't type the 10 special characters (such as dollar sign and
35: backslash) except as directed!  The following seven are printed by
36: typing a backslash in front of them:  \$  \&  \#  \%  \_  \{  and  \}.
37: The manual tells how to make other symbols.
38:
39: \end{document}                    % THE INPUT FILE ENDS LIKE THIS
```

Figure 7.1: A Sample LaTeX File

Simple Text

Words are separated by one or more spaces. Paragraphs are separated by one or more blank lines. The output is not affected by adding extra spaces or extra blank lines to the input file.
Double quotes are typed like this: "quoted text". Single quotes are typed like this: 'single-quoted text'.
Long dashes are typed as three dash characters—like this.
Italic text is typed like this: *this is italic text*. Bold text is typed like this: **this is bold text**.

A Warning or Two

If you get too much space after a mid-sentence period—abbreviations like etc. are the common culprits)—then type a backslash followed by a space after the period, as in this sentence.
Remember, don't type the 10 special characters (such as dollar sign and backslash) except as directed! The following seven are printed by typing a backslash in front of them: $ & # % _ { and }. The manual tells how to make other symbols.

Figure 7.2: The result of processing the sample file

7.2 Step 2. Packages, Macros and Sectional units

Packages and Macros

While you think about the structure of your document, you will probably find that there are some areas where basic LaTeX cannot solve your problem. If you want to include graphics, colored text or source code from a file into your document, you need to enhance the capabilities of LaTeX. Several prepared macros or enhancements are available for LaTeX—they are called packages. Packages are activated with the following command:

```
\usepackage[options]{package}
```

Package is the name of the package and *[options]* is a list of keywords which trigger special features in the package. Some packages come with the LaTeX 2_ε base distribution, others are freely available on the Internet.

Sectional Units

Documents, like this one, are often divided into sections. Each section has a heading containing a title and a number for easy reference. LaTeX has a series of commands that will allow you to identify different sorts of sections. All the sectioning commands take the section title as their argument (see lines 13 and 28) in Figure 7.1.

The most common sectioning commands are:

```
\chapter        \subsection
\section        \subsubsection
```

Table 7.2: Document Class Options

`10pt`, `11pt`, `12pt` Sets the size of the main font for the document. If no option is specified, `10pt` is assumed. `10pt` is TOO SMALL for letter size paper. Use a larger one.
`fleqn` Typesets displayed formulae left-aligned instead of centered.
`leqno` Places the numbering of formulae on the left hand side instead of the right.
`titlepage`, `notitlepage` Specifies whether a new page should be started after the document title or not. The `article` class does not start a new page by default.
`twoside, oneside` Specifies whether double or single sided output should be generated. The classes `article` and `report` are single sided by default.

`\chapter` is not available in the documentclass `article`. The commands should be used in the order given, since sections are numbered within chapters, subsections within sections, etc. Of course the spacing between sections, the numbering and the font size of the titles will be set automatically by LaTeX.

7.3 Step 3. Prepare your input file

There are some LaTeX 2_ε commands that must appear in every document. The actual text of the document always starts with a `\begin{document}` command and ends with an `\end{document}` command (see lines 10 and 39). Anything that comes after the `\end{document}` command is ignored. Everything that comes before the `\begin{document}` command is called the *preamble*. The preamble can only contain LaTeX commands to describe the document's style.

One command that must appear in the preamble is the `\documentclass` command (see line 9). This specifies what sort of document you intend to write (your choice in Step 1). After that, you can include macro commands which influence the style of the whole document or you can load packages which add new features to the LaTeX system (step 2).

Now that you know the basic stuff from which a LaTeX 2_ε document is made, this section will fill you in on the commands, environments, and modes you will need to know in order to produce the 'meat' of the input—the stuff between the `\begin{document}` and `\end{document}` commands.

Many non-technical documents consist almost entirely of running text—words formed into sentences, which are in turn formed into paragraphs—and the example file is no exception. Describing running text poses no problems, you just type it in naturally. In the output that it produces, LaTeX will fill lines and adjust the spacing between words to give tidy left and right margins. The spacing and distribution of the words in your input file will have no effect at all on the eventual output. Any number of spaces in your input file are treated as a single space by LaTeX, it also regards the end of each line as a space between words (see lines 15–17). A new paragraph is indicated by a blank line in your input file, so don't leave any blank lines unless you really wish to start a new paragraph.

LaTeX reserves a number of the less common keyboard characters for its own use. The ten characters

```
# $ % & ~ _ ^ \ { }
```

should not appear as part of your text. If you enter them into your text directly, they will normally not print, but rather coerce LaTeX into doing things you did not intend. All of these characters can be used in your documents by adding a prefix backslash. The exception is the backslash character itself which can *not* be entered by adding another backslash in front of it (\\) as this sequence is used for line breaking.[1]

Quotation Marks

LaTeX can print both opening and closing quote characters, and can manage either of these either single or double. To do this it uses the two quote characters from your keyboard: ‘ and ’. You will probably think of ’ as the ordinary single quote character which probably looks like ´ or ' on your keyboard, and ‘ as a "funny" character that probably appears as `. You type these characters once for single quote (see line 21), and twice for double quotes (see line 20). The double quote character " itself is never used.

For quotation marks you should *not* use the " as on a typewriter. In publishing there are special opening and closing quotation marks. In LaTeX, use two ‘s for opening quotation marks and two ’s for closing quotation marks.

Accents and Special Characters

LaTeX supports the use of accents and special characters from many languages.

```
H\^otel, na\"\i ve, \'el\‘eve \\
Comment \c{c}a?
```

Hôtel, naïve, élève
Comment ça?

7.3.1 LaTeX commands

LaTeX commands are case sensitive and take one of the following two formats:

[1] Try the $\backslash$ command instead. It produces a '\'.

- They start with a backslash \ and then have a name consisting only of letters. Command names are terminated by a space, a number or any other 'non-letter'.
- They consist of a backslash and exactly one numerical or special character.

LaTeX ignores whitespace after commands. If you want to get a space after a command, you have to put either {} and a blank or a special spacing command after the command name. The {} stops LaTeX from eating up all the space after the command name.

We have already seen some commands in the sample document, see lines (9, 10, 25, ...). Here are a couple more examples

```
Footnotes\footnote{This
  is a footnote} are often used
by people using \LaTeX.
```

Footnotes[a] are often used by people using LaTeX.

[a] This is a footnote

In manuscripts produced by typewriter, important words are underlined. In printed books these words are *emphasized*. The command to switch to an *emphasized* font is called:

```
\emph{text to be emphasized}
```

Here is a list of some of the other available typefaces: remember, the text to be acted upon must be inside a pair of braces to limit the amount of text that is affected.

`\textrm`	Roman	`\textit`	*Italic*	`\textsc`	Small Caps
`\emph`	*Emphatic*	`\textsl`	*Slanted*	`\texttt`	`Typewriter`
`\textbf`	**Boldface**	`\textsf`	Sans Serif		

7.3.2 Environments

To typeset special purpose text, LaTeX defines many different environments for all sorts of formatting:

`begin{`*name*`}` *text* `end{`*name*`}`

Here *name* is the name of the environment. Environments can be called several times within each other as long as the calling order is maintained.

```
\begin{aaa}...\begin{bbb}...\end{bbb}...\end{aaa}
```

In the following sections several environments are demonstrated.

7.4 Steps 4-6, Compiling, Error messages and previewing

Once you have completed typing your input file you may submit it to the compiler for compilation of a .dvi file. You can run LaTeX by typing

```
\flushleft
\begin{enumerate}
\item You can mix the list
environments to your taste:
\begin{itemize}
\item But it might start to
look silly.
\item[-] With a dash.
\end{itemize}
\item Therefore remember:
\begin{description}
\item[Stupid] things will not
become smart because they are
in a list.
\item[Smart] things though, can be
presented beautifully in a list.
\end{description}
\end{enumerate}
```

1. You can mix the list environments to your taste:
   - But it might start to look silly.
   - With a dash.
2. Therefore remember:

   Stupid things will not become smart because they are in a list.

   Smart things though, can be presented beautifully in a list.

```
\begin{flushleft}
This text is\\ left aligned.
\LaTeX{} is not trying to make
each line the same length.
\end{flushleft}
```

This text is
left aligned. LaTeX is not trying to make each line the same length.

```
\begin{flushright}
This text is right\\ aligned.
\LaTeX{} is not trying to make
each line the same length.
\end{flushright}
```

This text is right
aligned. LaTeX is not trying to make each line the same length.

```
\begin{center}
At the center\\of the earth
\end{center}
```

At the center
of the earth

```
qed:\home\parkinss>latex myfile.tex
```

where myfile.tex is the name of the LaTeX input file you have just created. You will see something similar to the following displayed.

```
This is TeX, Version 3.14159 (Web2C 7.2)
(myfile.tex
LaTeX2e <1997/12/01> patch level 2
Babel <v3.6j> and hyphenation patterns for american, french, german, nohyphenat
ion, loaded.
(/usr/lib/texmf/tex/latex/base/article.cls
```

```
Document Class: article 1997/10/10 v1.3x Standard LaTeX document class
(/usr/lib/texmf/tex/latex/base/size10.clo))
No file myfile.aux.
[1] (myfile.aux) )
Output written on myfile.dvi (1 page, 232 bytes).
Transcript written on myfile.log.
```

After running LaTeX , the formatted output of the program will be written to a file named myfile.dvi. In general, LaTeX will place the formatted output into a file with the same name as the .tex file, except with the .tex extension replaced by .dvi. The only way to view the dvi file is by printing it, or by using a dvi file previewer such as xdvi.

By now it should be clear that we have to work quite accurately when preparing a document. Spelling mistakes and typing errors can be absorbed, but messing up a control sequence name will halt the compiler with an error message.

LaTeX error messages appear frightening at first sight, to say the least. They are very informative but can take some getting used to. Mistyped control sequences aren't too bad but a missing `\end{environment}` can cause a great deal of confusion because it has the effect of making LaTeX try to set material into that environment that was never intended for such a place.

Suppose we type `\textbold` instead of `\textbf` in the following line:

```
this is going to be \textbold{very} messy.
```

This produces the following error message.

```
! Undefined control sequence.
l.683 this is going to be \textbold
                                  { very} messy.
?
```

That's not so bad. The line beginning with ! tells us that we have tried to use a control sequence that was not known to LaTeX; the 1.683 tells us that the error occurred on line 683 of the source file; and the error message is split over two lines with the break at the point that LaTeX detected a problem.

LaTeX error messages aren't too bad once you've made enough errors to get used to a few! Most can be avoided through careful preparation of the source file. Typing accurately and knowledge of the command syntax is a good start, but there are some other precautions that make good sense:

1. Use Emacs! The AUCTeX package in Emacs makes your life a whole lot easier (it will handle 2 and 3 below). Look at section 6.10.2 in Chapter 6.

2. Even if LaTeX is happy with free-form input, try to lay your input file out as regularly and logically as possible. See our examples of environments for formats to adopt.

3. It is important that all group delimiters be properly matched, i.e., braces and `\begin{}` ... `\end{}` must come in pairs. A good habit to fall into is to always type such things in pairs and then move the cursor back between them and type the intervening material.

4. Don't forget command arguments when they are mandatory. Always ask yourself what a particular command needs from you in order to make the decisions that are required of it.

5. Remember the characters that are specially reserved for commenting, table item separation, etc.

6. When we look at mathematical typesetting in the next chapter, we will see that the same principles apply there as well.

7.4.1 Previewing your document

The file previewers mentioned here only work within the X Windows environment.

You can check the formatting of the document without having to print it. To do this, there is a dvi file previewer named xdvi. To view a dvi file using xdvi type:

```
qed:\home\parkinss>xdvi myfile.dvi
```

which will bring up a window with the formatted document displayed in it. Xdvi has buttons on the side for you to change magnification, go forward or backward through the document, and to quit the program.

7.4.2 Printing your document from qed

Provided there is paper in the printer the command

```
qed:\home\parkinss>lpr myfile.dvi
```

will print your LaTeX document.

7.5 Wrap-up

We have learned pretty much all we need to know in order to prepare non-mathematical documents. There has been quite a lot of material, all told, but we're fortunate that many documents require only a fraction of what we've listed here. Furthermore, we'll find that what we've learned equips us with a good deal of the framework needed for mathematical typesetting.

I cannot emphasize enough the importance of getting your mind out of "word-processing" mode and into "typesetting" mode. Always keep in your mind the task at hand: describe the the logical content of the document to LaTeX, furnishing it with enough information to do the formatting for you.

We must recognize that there is a lot more to some of the commands than detailed here. Consult a LaTeX manual for more.

7.5.1 Typesetting Mathematical Formulae – Math Mode

In the present section, we'll learn how LaTeX typesets mathematics. It should come as no surprise that LaTeX does most of the work for us.

In text-only documents we saw that our task was to describe the logical components of each sentence, paragraph, section, etc. When we tell LaTeX to go into mathematical mode, we have to describe the logical parts of a formula, operator, special symbol, etc. TeX has been taught to recognize a binary operation, a variable, an operator that expects limits, and so on. We just need to supply the parts that make up each of these, and TeX will take care of the rest. When you want to revert to setting normal text again, you tell LaTeX to leave math mode and go back into the mode it was in (paragraphing mode).

LaTeX cannot be expected to perform these mode shifts itself, for it is not always clear when it is mathematics that has been typed. For example, should an isolated letter 'a' in the input file be regarded as a word (as in the definite article) or a mathematical variable (as in the variable 'a'). There are no reliable rules for LaTeX to make such decisions by, so the begin-math and end-math mode switching is left entirely to you.

Mathematical text within a paragraph is entered between `$` and `$` or between `\begin{math}` and `\end{math}`.

```
Add $a$ squared and $b$ squared
to get $c$ squared. Or using
a more mathematical approach:
$c^{2}=a^{2}+b^{2}$
```

Add a squared and b squared to get c squared. Or using a more mathematical approach: $c^2 = a^2 + b^2$

It is preferable to *display* larger mathematical equations or formulae, that is to typeset them on separate lines. To do this you enclose the equation between `$$` and `$$` or between `\begin{displaymath}` and `\end{displaymath}`. This produces formulae which are not numbered. If you want LaTeX to number them, you can use the equation environment.

```
Add $a$ squared and $b$ squared
to get $c$ squared. Or using
a more mathematical approach:
\begin{displaymath}
c^{2}=a^{2}+b^{2}
\end{displaymath}
And just one more line.
```

Add a squared and b squared to get c squared. Or using a more mathematical approach:

$$c^2 = a^2 + b^2$$

And just one more line.

There are differences between *math mode* and *text mode*. For example, in *math mode*:

1. Most spaces and line breaks do not have any significance, as all spaces are either derived

logically from the mathematical expressions or have to be specified using special commands such as `,`, `quad` or `qquad`.

2. Empty lines are not allowed. Only one paragraph per formula.

```
\begin{equation}
\forall x \in \mathbf{R}:
\qquad x^{2} \geq 0
\end{equation}
```

$$\forall x \in \mathbf{R}: \qquad x^2 \geq 0 \tag{7.1}$$

Most math mode commands act only on the next character. So if you want several characters affected by a command you have to group them together using curly braces: `{...}`.

```
\begin{equation}
Y = K^{\alpha + \theta}L^{1-\alpha-\theta}
\end{equation}
```

$$Y = K^{\alpha+\theta}L^{1-\alpha-\theta} \tag{7.2}$$

Greek letters are obtained by preceding the name of the letter with a backslash. To obtain upper case Greek letters the first letter of the letter name should be in uppercase. So `\alpha`, `\beta`, `\gamma` give you lower case Greek letters and uppercase letters are entered as `\Gamma`, `\Delta`, ...

```
$\lambda,\xi,\pi,\mu,\Phi,\Omega$
```

$\lambda, \xi, \pi, \mu, \Phi, \Omega$

Exponents and Subscripts can be specified using the `^` and the `_` character.

```
$a_{1}$ \qquad $x^{2}$ \qquad
$e^{-\alpha t}$ \qquad
$a^{3}_{ij}$
```

$a_1 \qquad x^2 \qquad e^{-\alpha t} \qquad a^3_{ij}$

Names of many standard functions are often typeset in an upright font and not italic as variables. Therefore LaTeX supplies commands to typeset function names. Here are a few:

```
\log     \ln     \exp   \min   \max     \Pr     \inf   \sup   \cos   \sin
```

```
\[U=\ln(c_1)+\ln(c_2)\]
```

$$U = \ln(c_1) + \ln(c_2)$$

A built-up **fraction** is typeset with the `frac{...}{...}` command. Often the slashed form 1/2 is preferable, because it looks better for small amounts of 'fraction material.'

The **integral operator** is generated with `int`, the **sum operator** with `sum`. The upper and lower limits are specified with `^` and `_` as with subscripts and superscripts.

If you put the command `left` in front of an opening delimiter or `right` in front of a closing delimiter, TeX will automatically determine the correct size of the delimiter. Note, that you must close every `left` with a corresponding `right`.

```
$1\frac{1}{2}$~hours
\begin{displaymath}
\frac{ x^{2} }{ k+1 }\qquad
x^{ \frac{2}{k+1} }\qquad
x^{ 1/2 }
\end{displaymath}
```

$1\frac{1}{2}$ hours

$$\frac{x^2}{k+1} \qquad x^{\frac{2}{k+1}} \qquad x^{1/2}$$

```
\begin{displaymath}
\sum_{i=1}^{n} \qquad
\int_{0}^{\frac{\pi}{2}} \qquad
\end{displaymath}
```

$$\sum_{i=1}^{n} \qquad \int_{0}^{\frac{\pi}{2}}$$

```
\begin{displaymath}
U^{'}\left(c_1\right)=\beta
E\left( \left( 1+r \right)
U^{'}\left(c_2\right)\right)
\end{displaymath}
```

$$U^{'}\left(c_1\right)=\beta E\left(\left(1+r\right)U^{'}\left(c_2\right)\right)$$

To enter **three dots** into a formula you can use several commands. `ldots` typesets the dots on the baseline, `cdots` sets them centered. Besides that there are the commands `vdots` for vertical and `ddots` for diagonal dots.

```
\begin{displaymath}
x_{1},\ldots,x_{n} \qquad
x_{1}+\cdots+x_{n}
\end{displaymath}
```

$$x_{1},\ldots,x_{n} \qquad x_{1}+\cdots+x_{n}$$

For More Information

A more comprehensive introduction to LaTeX can be found in *The Not So Short Introduction to LaTeX 2ε* available in Postscript or pdf format in /usr/doc/qed on qed.

Chapter 8

Electronic Mail with Pine

Pine is a registered trademark of the University of Washington.

Pine is an electronic messaging program created and maintained by the Computing & Communications group at the University of Washington. This Chapter is based on the document "Getting Started with Pine" a 1997 document prepared by the University of Washington Computing & Communications. The original document is available at `http://www.washingon.edu/pine/tutorial/`

Starting Pine

To start: type pine as a command at the Unix system prompt. After starting Pine, the Main Menu screen appears. Each Pine screen has a similar layout: the top line tells you the screen name and additional useful information, below that is the work area (on the Main Menu screen, the work area is a menu of options), then the message/prompt line, and finally the menu of commands.

To quit: When you want to leave Pine, press [q] (Quit).

The Main Menu

The Main Menu lists Pine's main options (see Figure8.1). The key or keys you must type to enter your choice is to the left of each option or command name. You can usually type either uppercase or lowercase letters, and you should not press [return] to enter commands.

From the Main Menu you can choose to read online help, write (compose) and send a message, look at an index of your mail messages, open or maintain your mail folders, update your address book, configure or update Pine, and quit Pine. There are additional options listed at the bottom of the screen as well.

Now that you know how to start Pine, you can explore on your own, or you can browse the rest of this document for a summary of Pine's main features.

Figure 8.1: A Pine Main Menu Screen

Getting Help in Pine

To read the online help, use the Help command at the bottom of each screen. For example, at the Main Menu screen, press ? (Help). Because the help text is context sensitive, you never see all of it at once-only the part that relates to the Pine feature you are using. To exit the online help, press e (Exit Help).

8.1 Writing a Message in Pine

To write a message, press c (Compose). You see the Compose Message screen.

In the command menu above, the circumflex character is used to indicate the Control key. This character means you must hold down the Control key (written in this document as Ctrl) while you press the letter for each command.

Different commands are available to you when your cursor is in different fields on this screen. To see additional commands available when your cursor is in the Message Text field, type Ctrl-g (Get Help). For example, to move around, use the arrow keys or Ctrl-n (Next line) and Ctrl-p (Previous line); to correct typing errors, Backspace or Delete.

Figure 8.2: A Pine Compose Message Screen

You might start experimenting in Pine by sending yourself a message. The following section shows you how.

Writing and Sending a Test Message to Yourself

To write and send a test message to yourself:

1. Press C (Compose) to see the Compose Message screen.
2. In the To field, type your email address and press return.
3. In the Cc field, press return.
4. In the Attachment field, press return.
5. In the Subject field, type Test and press return.
6. Below the Message Text line, type This is a test.

If a user whose userid is *parkinss* at site *qed.econ.queensu.ca* were to compose such a test message, the completed screen would look like the following example.

Figure 8.3: A Pine Folder Index Screen

7. To send your message, type Ctrl-x

You are asked:

Send message?

8. Press y (yes) or press return.

The message is sent and a copy is saved to your sent-mail folder. (If you press n (no) the message is not sent, and you can continue to work on it.)

You have just sent a basic message. There are, of course, other options you can use as you compose a message. A few are summarized in the next section, and complete information about options for the Compose Message screen is available in Pine's online help. As you compose a message, you can type Ctrl-g (Get Help) at any time to see details about your current task.

Hints for Writing a Message

To: In this field, type the email addresses of your recipients. Separate the addresses with commas. When you are finished, press return. Always check the addresses in both the To and the Cc fields for accuracy and completeness before you send a message.

Cc: In this field, type the email addresses of the persons to whom you want to send copies. Separate their addresses with commas. If you need more lines Ctrl-r will give you more. When you are finished, or if you do not want to send any copies, press return.

Attchmnt: This is an advanced Pine feature that allows you to attach files, including word processing documents, spreadsheets, or images that exist on the same computer where you are running Pine. If you do not want to attach a file to your message, press return. For more information, place your cursor in the Attchmnt field, then type Ctrl-g (Get Help).

Subject: In this field, enter a one-line description of your message. A short, pertinent description is appreciated by recipients, since this is what they see when they scan their index of messages. When finished, press return.

Message Text: Type your message. To move around, use the arrow keys. To delete a character, press Backspace or delete. To delete a line, type Ctrl-k. To justify text, type Ctrl-j. (To immediately undelete a line or to unjustify text, type Ctrl-u). To check the spelling, type Ctrl-t. To see other editing commands, type Ctrl-g (Get Help).

Hints for Sending a Message

Sending a Message. After your message is composed, type Ctrl-x, and then press y or press return. Your message is sent and a copy is saved to the sent-mail folder. If a message cannot be delivered, it eventually is returned to you. If you want to re-send a message, you can use the F (Forward) command.

Changing Your Mind. If you change your mind after typing Ctrl-x to send a message, press n

instead of [y] to continue to work on your message. While you are writing your message, you can type [Ctrl-o] (Postpone) to hold your message so you can work on it later, or you can type [Ctrl-c] (Cancel) to delete your message entirely. You are asked to confirm whether or not you want to cancel a message.

8.2 Listing, Viewing, Replying to, and Forwarding Messages

Pine stores messages that are sent to you in your INBOX folder. Messages remain in your INBOX until you delete them or save them in other folders.

Listing Messages

To see a list of the messages you have received in your INBOX folder:

At the Pine Main Menu, press [i] (Folder Index).

The selected message is highlighted. The first column on the left is blank, or shows a "+" if the message was sent directly to you (i.e., it is not a copy or from a list).

The second column may be blank, or it may contain: "N" if the message is new (unread), "A" if you have answered the message (using the Reply command), "D" if you have marked the message for deletion.

The rest of the columns in the message line show you the message number, date sent, sender, size, and subject. For details, press [?] (Help).

Most of the commands you need to handle your messages are visible at the bottom of the screen, and you can press [o] (OTHER CMDS) to see additional commands that are available. You do not need to see these "other commands" on the screen to use them. That is, you never need to press [o] as a prefix for any other command.

Viewing a Message

To view a message:

1. At the Folder Index screen, use the arrow keys to highlight the message you want to view.
2. Press [v] (ViewMsg) or press [return] to read a selected message.

To see the next message, press [n] (NextMsg).

To see the previous message, press [p] (PrevMsg) To return to the index, press [i] (Index).

Replying to a Message

To reply to a message that you have selected at the Folder Index screen or that you are viewing:

Press [r] (Reply).

You are asked whether you want to include the original message in your reply. Also, if the original message was sent to more than one person, you are asked if you want to reply to all recipients. Think carefully before you answer-it may be that you do not want your reply to be sent to more than just the author of the message.

It is always a good idea to check the list of addresses in the To and Cc fields before you send a message to see who will receive it.

Forwarding a Message

To forward a message that you have selected at the Folder Index screen or that you are viewing:

1. Press [f] (Forward).

A copy of the message opens and the To field is highlighted.

2. Enter the address of your recipient and send the message as usual. Note that you can modify the original message if you wish, for example, to forward only a portion of it or to add a message or notes of your own.

8.3 Pine Folders

Messages can quickly accumulate in your INBOX folder. If you use email often, you soon could have hundreds. You need to delete messages you do not want, and you can use folders to organize messages you wish to save. A folder is a collection of one or more messages that are stored (just like the messages in your INBOX) so you can access and manage them.

Organizing Messages With Folders

You can organize your email messages into different folders by topic, correspondent, date, or any other category that is meaningful to you. You can create your own folders, and Pine automatically provides three:

- The INBOX folder-messages sent to you are listed in this folder. When you first start Pine and go to the Folder Index screen, you are looking at the list of messages in your INBOX folder. Every incoming message remains in your INBOX until you delete it or save it in another folder.
- The saved-messages folder-copies of messages you save are stored in this folder unless you save them to other folders you create yourself.
- The sent-mail folder-copies of messages you send are stored in this folder. This is convenient if you cannot remember whether you actually sent a message and want to check, or if you want to send a message again.

Moving Between Folders

When you start Pine and press [i] (Index) at the Main Menu, you see a list of messages in your INBOX folder. If you want to see the messages in another folder, you need to go to that folder. The following text shows you two ways to go to another folder from nearly anywhere in Pine.

To access your folders and the messages that are stored in them:

1. Press [l] (ListFldrs). You see the Folder List screen with collections of folders. Typically each collection is shown as in the example below with a "Select Here to See Expanded List" button below each.

2. If it is not already highlighted, use the arrow keys to highlight the "Select Here to See Expanded List" button below the desired collection (e.g., Folder-collection of mail) and press [return].

You see an expanded list of folders, similar to the following, in which your current folder is highlighted.

3. Use the arrow keys to highlight another folder.

4. To see an index of the messages in that folder, press [v] (ViewFldr) or press [return].

To move most quickly to the index of another folder:

1. From almost anywhere in Pine, press [g] (GotoFldr). You are prompted for the name of a folder.

2. Type the folder name.

3. Press [return]. You see the list of messages in the folder.

8.4 Saving a Message

When you save a message, you are given a choice: you can store it in the saved-messages folder, or you can specify another folder.

Once you save a message, the copy in the INBOX folder automatically is marked for deletion so that you only will have one copy. When you quit Pine, you are asked to confirm whether or not you want to expunge the copy from the INBOX folder. To conserve space, it is a good idea to do this.

Saving a Message to the Saved-Messages Folder

To save a message to the saved-messages folder:

1. At the Folder Index screen, use the arrow keys to highlight the message you want to save, or, at the Message Text screen as you view a message:

Press [s] (Save).

You are asked if you want to save the message to the saved-messages folder or to another folder:

SAVE to folder in [saved-messages]: 2. Press [return] to choose the default folder: [saved-messages].

Pine saves your message, and you see the following:

[Message # copied to "saved-messages" and deleted]

Saving a Message to a Folder You Specify

You will find it useful to create additional folders for storing messages on particular subjects.

To save a message to a folder you specify:

1. At the Folder Index screen, use the arrow keys to highlight the message you want to save or, at the Message Text screen as you view a message:

Press [s] (Save) to save a message.

You are asked if you want to save it to the saved-messages folder or to another folder:

SAVE to folder in [saved-messages]: 2. Type a *foldername* and press [return].

For example, to save a message to a folder named "papers" type papers and press [return].

If this is the first time you have named this folder, you see the message:

Folder "papers" doesn't exist. Create?

Press [y] or press [return] to create the folder.

Once you have created the folder, or whenever you type the name of a folder that already exists, you see a message like this one:

[Message # copied to "papers" and deleted]

8.5 Deleting a Message

You keep your Pine folders clean by routinely deleting messages you do not want. There are two steps to deleting a message: marking it for deletion and then expunging it.

To mark a message you do not want for deletion:

1. Select and open the folder that contains the message you wish to mark for deletion.

2. At the Folder Index screen, select the message you want to mark for deletion, or simply view the message.

3. Press [d] (Delete).

If you are looking at the Folder Index screen when you mark a message for deletion, a "D" appears in the left column of the message line, and the next message, if there is one, is selected.

If you are looking at the Message Text screen when you mark a message for deletion, a "DEL" briefly appears in the upper right corner of your screen, you get an on-screen message that the message has been deleted, and the next message, if there is one, appears.

Repeat this process to mark additional messages for deletion.

*

Expunging a Message A message that is marked for deletion remains in Pine until you expunge it. You can expunge a message that is marked for deletion at any time, or you can wait until you quit Pine. Once you have a few messages marked for deletion, you may want to expunge them before you continue to work, because it is easier to look through a folder index that contains fewer messages.

To expunge a message:

1. At the Folder Index screen, press [x] (expunge). You are asked: Expunge # message(s) from "*foldername*"? 2. Press [y] (yes) or press [return]. Messages marked for deletion disappear.

[Note: You will be asked whether you want to expunge messages that are marked for deletion whenever you leave a folder (other than the INBOX) that contains messages marked for deletion, or when you quit your Pine session]

8.6 Using the Address Book

As you use email, you build a list of email correspondents. Some of their addresses may be difficult to type or remember. Pine provides an address book to make it easier to handle email addresses. At the Pine Main Menu, press [a] to see the Address Book screen. You can use the address book to store email addresses for individuals or groups, to create easily remembered "nicknames" for these addresses, and to quickly retrieve an email address when you are composing a message. Here is a sample page from an address book:

There are two ways to set up addresses in your address book; you can add them manually or take them from messages. With either method, you specify nicknames for your correspondents. A single address book entry (or nickname) can point to just one email address, or, it can point to more than one. When it points to more than one, it is called a distribution list. Each distribution list has a nickname, a full name, and a list of addresses. These addresses may be actual addresses, other nicknames in your address book, or other distribution lists.

There are many distribution lists available automatically on qed. These include `qed`, `mas`, `phds`, `students`. These distribution lists allow you to send mail to many people at once. DO NOT ABUSE THIS FACILITY. Mass mailings of 'stupid' things (like a fake trojan horse) is a serious offense which could be punished by loss of e-mail privileges.

Adding Single Addresses or Distribution Lists Manually

To add single addresses or distribution lists manually:

1. Have ready the address or addresses you want to add.

2. At the Pine Main Menu, press [a] (AddrBook). You see the Address Book screen.

3. Press [a] (AddNew) and follow the instructions. (Type [Ctrl-g] if you specifically need help to add a new address.)

Taking Single Addresses

To take a single address from a message you are viewing or have selected in the index:

1. At the Message Text or the Folder Index screen, press [t] (TakeAddr). [Note: The T command is not visible on your screen unless you press [o] (OTHER CMDS), but you need not see this command to use it.]

You see the Take Address screen.

If there is more than one address to take, you see this message:

Use P (Prev), N (Next), or the up and down arrow keys to select the address you want, and press [t] (Take).

At this point, or, if there is only one address to take, you see this message:

Enter new or existing nickname (one word and easy to remember): 2. Enter a nickname for your correspondent and press [return].

3. Follow the instructions. (Type [Ctrl-g] if you need help.)

Using Address Book Entries When Composing Email

When composing a message, at the To or the Cc (Carbon Copy) fields you can enter an email address in any of the following ways:

- Type the entire email address.
- Type a nickname you have set up in the address book.

Place your cursor in the To or Cc field, then type [Ctrl-t] (To AddrBk) and use the arrow keys to highlight the name you want. Press [s] (Select) or press [return].

8.7 Guidelines and tips for Using Email

Electronic mail is a unique medium of communication. Messages can be replied to or forwarded with speed and ease, and email has the potential to reach a wide audience. These features can also

be misused. There are a few basic guidelines for the responsible use of email that can help you avoid common mistakes while you enjoy the full benefits of this technology.

The privacy of an email message cannot be guaranteed. An email message may be forwarded, printed, or permanently stored by any recipient. Email can be misdirected, even when you are careful. Do not put something in an email message that you would not want read by everybody. And if you receive a message intended for someone else, let the sender know.

Email does not show the subtleties of voice or body language. Avoid attempts at irony or sarcasm. The most effective email is short, clear, and relevant. If you receive a message that makes you upset, do not respond immediately, and in any case, avoid "flaming," that is, sending an angry or rude message.

Email Tips

As you use email, keep the following tips in mind:

- Email is easily forwarded to someone else. Although this is convenient, it is not always appropriate. If you are unsure, ask the sender before you forward a message.
- Email replies may go to more people than you realize. When replying to a message be sure to look at the list of recipients, especially addresses of mailing lists, which may redistribute your message to dozens or hundreds of individuals.
- Email can be junk mail, so avoid unnecessary proliferation of messages.
- Email takes up computer space, so delete messages you no longer need.
- The integrity of an email message cannot be guaranteed. If a received message seems out of character for the sender, double-check before taking it seriously.
- Email is meant for informal correspondence as well as scholarly, scientific, and clinical communications. You should not use email for official record purposes where a memo would be required (e.g., personnel actions, organization changes, contracts, and policy statements).
- Email should not be considered private. Confidential information should not be sent by email.

8.8 Quitting Pine

1. At almost any place in Pine, press q (Quit). You are asked: Really quit pine? 2. Press y (yes) or press return to quit.

Chapter 9

Using remote systems

Unix operating systems are very good at networking. In this chapter we'll look at programs to connect to and exchange files with other computers.

9.1 Using Systems by Remote

`telnet` *remote-system*

A common way of using a remote Unix system is through `telnet`. `telnet` is a fairly simple program to use:

```
qed:/home/parkinss telnet fraser.sfu.ca
Trying
Connected to fraser
Escape character is '^]'.

fraser login:
```

As you can see, after I issue a `telnet` command along with a machine name, I'm presented with a login prompt for the remote system. I can enter my username (and password) and use the remote system almost as if I were sitting at its console.

The normal way of exiting `telnet` is to `exit` on the remote system.

9.2 Exchanging Files

`ftp` *remote-system*

The most common way of sending files between Unix systems is `ftp`, for the **file transfer protocol**. After entering the `ftp` command, you'll be asked to login to the remote system, much like `telnet`. After doing so, you'll get a special prompt: an `ftp` prompt.

The `cd` command works as normal, but on the remote system: it changes your directory on the *other* system. Likewise, the `ls` command will list your files on the remote system.

The two most important commands are `get` and `put`. `get` will transfer a file from the remote system locally, and `put` will take a file on the local system and put in on the remote one. Both commands work on the directory in which you started `ftp` locally and your current directory (which you could have changed through `cd`) remotely.

One common problem with `ftp` is the distinction between text and binary files. `ftp` is a very old protocol, and there used to be advantages to assuming that files being transferred are text files. Some versions of `ftp` default to this behavior, which means any programs that get sent or received will get corrupted. For safety, use the `binary` command before using `get` or `put`.

To exit `ftp` use the `bye` command.

9.3 Using ssh and scp

`ssh` *remote-system*

Most hosts in the `econ.queensu.ca` domain have been installed with a package called `ssh`. This software allows you to connect to the host machines using secure encrypted sessions.

Normally when you telnet to or from a host on the network the password that you type is sent in clear text. Anyone who has access to the network or networks between you and the host you are logging into can capture your password (with a sniffer) [1]. Only by using encryption can the threat of sniffing be stopped. `ssh` protects your password and it prevents someone else from seeing everything you do while you're remotely logged in.

`ssh` sessions also allow you easy access to X applications securely over remote logins. You don't need to set your DISPLAY environment variable or configure your X server authorization. After logging in to the host via `ssh` just enter the command for the X program you wish to start, and voila it appears on your local display.

The following sections hope to provide enough information to setup a user new to *ssh* with the appropriate files necessary for accessing remote hosts in a secure manner. See man pages for more.

[1] A *sniffer* is a program which watches all the traffic on the network to which a computer is attached. Someone installs a "sniffer" on a machine which looks for passwords on the network. Anyone who logs into a host on that network, or logs onto another host from that network then gets their password captured.

About public key cryptography

Public key cryptography uses a *public key* to encrypt data and a *private key* to decrypt it. The name *public key* comes from the fact that you can make the encryption key public without compromising the secrecy of the data or the decryption key.

What this means is that it is safe to send your *public key* (i.e. the contents of the ~/.ssh/identity.pub file) in electronic mail or by other means e.g. to have a system administrator of a remote site install that key into your ~/.ssh/authorized_keys file. For anyone to actually gain access they need the corresponding *private key* (i.e. the decrypted contents of ~/.ssh/identity) to identify themselves.

To further protect your *private key* you should enter a pass-phrase to encrypt the key when it is stored in the filesystem. This will prevent people from using it even if they gain access to your files.

Creating your authentication key

The very first step is to use ssh-keygen to create an authentication key for yourself. In most cases the defaults for this command are what you want.

Always, **always**, type in a good pass-phrase when prompted for one. It can be multiple words (i.e. spaces are just fine within the phrase), so you could choose a sentence that you can remember. Changing some of the words by misspelling them or by changing some of the letters into digits is highly recommended to increase the strength of your pass phrase.

Here is a sample session. Note that the pass-phrase is *not* echoed back as you type it. (You can change the pass-phrase at any time by using the -p option of ssh-keygen.)

```
jsp_fraser%  ssh-keygen
Initializing random number generator...
Generating p:  .++ (distance 6)
Generating q:  ........++ (distance 110)
Computing the keys...
Testing the keys...
Key generation complete.
Enter file in which to save the key (\$HOME/.ssh/identity): [Press Return]
Enter passphrase (empty for no passphrase): litt1e 1amp jumb3d
Enter same passphrase again:  litt1e 1amp jumb3d
Your identification has been saved in /home/jsp/.ssh/identity.
Your public key is:
1024 37 [lots of numbers] jsp@fraser.sfu.ca
Your public key has been saved in /home/jsp/.ssh/identity.pub
```

Authorizing access

To allow access to a system for a given identity place the *public* key in your ~/.ssh/authorized_keys file on that system. All keys listed in that file are allowed access.

Usually you will want to authorize access to the local system using the local key (especially in an environment where multiple systems share the same home directory e.g. using NFS). Thus a good start is to copy the public key for your default identity into the ~/.ssh/authorized_keys file.

```
jsp_fraser% cd ~/.ssh
jsp_fraser% cp identity.pub authorized_keys
```

You could now copy the ~/.ssh/authorized_keys file to other systems to allow access from the local system.

Use a text editor to add more keys to the file. If you use cut and paste to copy the key make sure each key entry is *a single line* in the file. The keys to add are *always* the public keys (from files with the .pub extension).

Logging into remote systems

To establish an interactive connection to a remote system you would use the ssh command. The `-l` option is used if your `userid` on the two machines differ.

```
jsp_fraser% ssh -l parkinss qed.econ.queensu.ca
Enter passphrase for RSA key 'jsp@fraser.sfu.ca': litt1e 1amp jumb3d
Last login: Tue Jun  9 16:33:03 1998 from fraser.sfu.ca
[message of the day from qed and mail waiting]

qed:/home/parkinss>
```

If you are using the X Window System you can use this capability to start a terminal window to start an interactive session on the remote system. Once I log into `qed` with `ssh` X programs I initiate on `qed` use the screen on my local terminal in Vancouver.

Copying files between systems

You can copy files from the local system to a remote system or vice versa, or even between two remote systems using the `scp` command. When possible use `scp` instead of `ftp`

To specify a file on a remote system simply prefix it with the name of the remote host followed by a colon.

`scp` file *user@remote-system:.*

If you leave off the filename of the copy or specify a directory only the name of the source file will be used. An easy way of retrieving a copy of a remote file into the current directory while keeping the name of the source file is to use a single dot as the destination.

```
jsp_fraser% scp -p parkinss@qed.econ.queensu.ca:todo .
jsp_fraser%
```

The -p option is not required. It indicates that the modification and access times as well as modes of the source file should be preserved on the copy. This is usually desirable.

Relative filenames resolve differently on the local system than on the remote system. On the local system the current directory is assumed (as usual with all commands). *On the remote system the command runs in the home directory!* Thus relative filenames will be relative to the home directory of the remote account.

Chapter 10

Econometric software

This is a very brief introduction to some of the commonly used software in the Queen's Economics Department. The Economics Department has site licenses for SHAZAM and TSP and that the University has a site license for Maple. Current students have a *right* to use the software. You are not required to pay a license fee to install these programs on your own computer.

10.1 SHAZAM

SHAZAM is an econometric software package used primarily by economists for both teaching and research. Its emphasis is on the statistical analysis of time series data (e.g. annual gross national product, monthly inflation rate). SHAZAM is available on **qed** for general research computing. The reference manual can be purchased in the Bookstore. Vicki should also have a copy.

Getting Started

Log on to qed and type

```
shazam
```

Remember that all letters must be in lower case. Once inside SHAZAM, you need not worry about upper and lower case except when referring to file names. The SHAZAM command prompt appears as

```
TYPE COMMAND
-
```

where the underscore (_) indicates the position of the cursor. For online help, type help. To terminate SHAZAM, type stop. The SHAZAM `file` command is useful for a variety of operations involving input and output. These are summarized below.

For example, to keep a record of your terminal session in a file called `project.lis` for subsequent printing, type the SHAZAM command

File command	Description
`file 11`	Attach unit 11 to a filename, perhaps to input data with a read command.
`file 12`	Attach unit 12 to a filename, perhaps to output data with a write command.
`file screen`	Used to copy to a file what subsequently appears on your screen.
`file input`	Used to input a file containing SHAZAM commands.

file screen project.lis

If this file does not already exist, it is created for you. If it does exist, any previous contents will be destroyed.

You can issue a Unix command while in SHAZAM by starting the command with a dollar ($) symbol. For example, to display the file names in your current directory in column format, type $ls -C

Suppose you have a file called `project.dat` containing two columns of data representing monthly rate of inflation and percentage unemployment. To input this data, type

```
file 11 project.dat
read (11) inf unemp
```

To regress unemployment on inflation, type

ols unemp inf

If you already have these commands in a file called **project.sha**, you would type

file input project.sha

To run SHAZAM in batch mode, type

```
shazam <test.sha > test.lis &
```

The < symbol tells SHAZAM to find the commands in a file named `test.sha`, and the > symbol tells SHAZAM to put the output in a file named `test.lis`. The & symbol puts the job in the background, allowing you to do other work until Unix notifies you that your job is completed.

Increasing Memory

SHAZAM requires all of your data in RAM to perform your analysis. Your data is organized internally as a rectangular array of numbers with rows representing observations and columns representing variables. Use the formula, rows x columns x 8 / 1024, to compute the minimum number of kilobytes (K) of memory required to store your data. Additional memory is required according to the type of analysis, so allow for overhead. The default is 1000K on qed. To request 2,000K of memory, for example, type the SHAZAM command **par 2000** before you input your data.

Quite often, data is organized with each variable stored as a stream of numbers in separate files (the Cansim series at CHASS for example). Although you can use units 11-49 for data files, you really need only one unit number to input your data. For example, type

```
file 11 inf.dat
read (11) inf / byvar
file 11 unemp.dat
read (11) unemp / byvar
```

By interlacing the file and read commands, you will not be restricted by the 39 available unit numbers. A similar strategy can be followed when creating a multitude of output files.

10.2 TSP

TSP, like SHAZAM, is an econometric software package whose emphasis is on the analysis of time series data. **Version 4.4** is available on **qed** for general research computing. The manual for TSP should be in Vicki's office.

Getting Started

Log on to qed and type

tsp

Remember that all letters must be in lower case. TSP will prompt you with

Enter batch filename [or press Enter for interactive]:

Press return for interactive use. Once inside TSP, you need not worry about upper and lower case except when referring to file names. The TSP command prompt appears as

1 ? _

where the underscore (_) indicates the position of the cursor. The command prompt number (e.g. 1 above) is incremented by one for each new command. Use the TSP `review` command to display the commands you have entered during the session. For online help, type help. To terminate TSP, type stop. Use the TSP `read` command to input your data in character format. Suppose you have a file named `project.dat` containing two columns of data,the first one for inflation and the second for unemployment. To input this data in free format, type

```
read (file='project.dat') inf unemp
```

The numbers must be separated by blanks or commas when reading the data in free format.

TSP commands cannot exceed 80 characters in length. For long commands, use the backslash (\) to continue a command onto a subsequent line. The TSP `save` and `restore` commands are used to keep your data in binary format between sessions. For example, the command `save mydata`

will create a file named `mydata.sav` in your current directory. In a subsequent session, type restore mydata to retrieve it and continue your work. If you do not specify a file name on the `save` and `restore` commands, the file name `TSPSAV.SAV` is used by default. To temporarily suspend a TSP session, type system to exit to Unix. You can then type any Unix command. To reinstate the TSP session, type exit.

The file named `BKUP.TSP` is created whenever you run TSP interactively. It contains a record of all the commands you typed during the session. Use the Unix `mv` command, for example

```
mv BKUP.TSP myfile.tsp
```

to rename the file and preserve its contents. This file is not altered when you run TSP in batch mode.

While in interactive mode, you can have your output directed to a file by typing, for example, `output 'test.out'`. However, the output does not appear on the screen as well. To overcome this deficiency, use the Unix **script** command prior to invoking TSP. For example, type

```
script test.out
tsp
(work interactively)
stop
exit
```

The above **script** command forces everything appearing on the screen to be recorded in the file named `test.out`. Recording is terminated with the Unix `exit` command.

How Can I Run TSP in Batch mode?

Suppose you have a file named `myfile.tsp` containing TSP commands as displayed below using the Unix `more` command:

```
more myfile.tsp
read (file='project.dat') inf unemp;
olsq unemp c inf;
```

Note that a semicolon is required at the end of each command when running TSP in batch mode. The `olsq` (ordinary least squares) command estimates the parameters of a regression model. The `c` indicates there is a constant or intercept term in the model.

To process these commands in batch mode, type

```
tsp myfile &
```

Note how the `.tsp` file extension is implied. The `&` puts the job in the background, allowing you to do other work while it is processed. Your output is placed in the file `myfile.out`.

Like SHAZAM, TSP requires all of your data in memory to perform your analysis. Your data is organized internally as a rectangular array of numbers with rows representing observations and

columns representing variables. The default memory is 4 megabytes. To request 8 megabytes of memory, for example, include the following as your first command

```
options memory=8;
```

10.3 LIMDEP

LIMDEP (LIMited DEPendent variables) is an econometric software package. Although it provides basic econometric techniques (e.g. two and three-stage least squares), its special emphasis is on modeling qualitative and limited dependent variables (e.g. logit and tobit models). In addition, LIMDEP can model panel data (i.e. time series data measured on a number of cross-sectional units). Version 7 is available on **qed** for general research computing. Vicki should have the manuals for running LIMDEP under Unix.

Getting Started

Log on to qed and type

```
limdep
```

Remember that all letters must be in lower case. Once inside LIMDEP, you need not worry about upper and lower case except when referring to file names. The LIMDEP command prompt appears as

```
Cmnd>
```

To terminate LIMDEP, type stop You will be prompted with

```
OK to Exit?(Work saved?)>
```

as a reminder to save your data. Answer `y` to quit, or `n` to resume execution.

For an interactive session, you will likely want to save your output in a file for printing afterwards. For example, type

```
open; output=printfile$
```

Most LIMDEP commands begin with a verb, end with a dollar symbol, and have one or more specifications separated by semicolons in between.

Suppose you have a file named `project.dat` containing two columns of data representing monthly rate of inflation and percentage unemployment. To input this data, type

```
read; nvar=2; names=inf,unemp;file=project.dat$
```

LIMDEP will respond with Unexpected END OF FILE at record n where `n` is one more than the number of lines in the file. This is normal for LIMDEP. To eliminate this message you would have to specify `;nobs=100`, for example, if there were 100 observations.

To display your data, type

```
list;*$
```

To regress unemployment on inflation, type

```
regress; lhs=unemp;rhs=one,inf$
```

To create a new variable which is the square of inflation type

```
create; inf2=inf*inf$
```

To save your input data and any newly created variables, in a binary file named `data.sys`, type

```
save; file=data.sys$
```

To restore this data in a subsequent session, type

```
load; file=data.sys$
```

If you already have your LIMDEP commands in a file named `project.lim`, for example, you could execute them by typing

```
open; input=project.lim$
```

When you run LIMDEP in batch mode you do not need the `open` statement. Type

```
limdep <project.lim> project.out &
```

10.4 STATA

Stata is a comprehensive and easy to learn statistical application which contains many of the statistical tools and models developed in the econometric literature. Currently, Stata version 5.0 is installed on qed.

Getting Started

You can execute Stata command files interactively or in batch mode. To use Stata in batch mode, type your command file as a text file using any Unix text editor such as Emacs and then submit the text file to Stata with a command in the form:

```
qed:/home/parkinss> stata -b do myfile.ext
```

where `myfile.ext` is the name of your command file with any extension `ext`. If you use the default extension `do`, you can ommit it from the command line. The results of your session will be located in the file `myfile.log`. For example, the command

```
qed:/home/parkinss> stata -b do job
```

tells Stata to execute the commands in the command file 'job.do', suppress all screen output, and route the output to a file named 'job.log'.

To start an interactive Stata session, type **stata** at the Unix prompt. Once you are in Stata, commands are issued at the prompt which is a period. You can exit Stata by typing either **exit** or **exit, clear**. The **exit** command is sufficient if all modifications made to the data during the session have been saved. Use **exit,clear** if you want to force Stata to exit without saving the modifications. By default, the Stata session is organized in the interactive mode. That is, each command is performed immediately after it has been typed and entered using return. The output will appear on the screen. The interactive mode may be convenient for simple exploratory analysis. To save the output generated during a Stata session, you should type a **log** command prior to any analysis performed. **log** commands have the following form:

```
. log using filename
```

where `filename` refers to the output file. By default, output files have the extension '.log '. For example, the command: *. log using job* will save all output generated after this command line into a file named 'job.log'. The option **append** can be used to add new output to a previously existing output file. For example, if the following line is typed at the beginning of a Stata session **. log using** job, **append** all new analyses will be saved to the file ' job.log ' at the end of the older output file. Alternately, the command `. log using job, replace` will replace the older 'job.log' file with the results of the analyses in the current Stata session.

Reading in RawData

Raw data stored in a separate ASCII data file, should be read in using the **infile** command which has the form

```
. infile variablelist using datafile
```

where `variablelist` is the list containing the names of the variables to be read by Stata from the data file. `datafile` is the full name of the ASCII file containing the data. Stata's default extension for raw data files is '.raw' . In other words, if you ask Stata to look for a raw data file called 'numbers', it will actually look for a file called 'numbers.raw'. You may use any other extension, but it will have to be explicitly specified as in the example below:

```
. infile id gender class grade using school.data
```

This command reads in four numeric variables from the file 'school.data'.

Changing the Amount of Memory Allocated to Data

Stata works with a copy of the data which it loads into memory. When you invoke Stata by typing **stata** at the Unix prompt, 1 megabyte is allocated to Stata's data areas. This amount may be insufficient when working with large data sets. The amount of memory allocated to data can be changed when invoking Stata by typing the following at the Unix prompt:

```
qed:/home/parkinss> stata -k \#
```

where # refers to the amount of memory requested. If #=2000 then stata allocates 2 megabytes of memory to data.

Stata's Commands

One of the main advantages of working with Stata compared to other general statistical software is the simple syntax of Stata's commands. All commands have the following form:

[**by**`varlist1` :] `commandname` [`varlist2`] [**if** `expression`] [**in**`range`] [**,***options*]

---

where the items enclosed in brackets are optional. *commandname* is a substitute for a key word signifying a performed procedure, for example **plot** or **ttest**. [**by***varlist1* :] repeats the command for each set of values of *varlist1*. Some commands allow this item, others do not. For example, the command

```
by gender  summarize income
```

reports the mean, variance, and other univariate statistics of the variable 'income' for men and women separately.

The command

```
regress income gender education  if region=1
```

performs a linear regression with the dependent variable 'income' and two independent variables 'gender' and 'education' where only those cases which have value '1' for the variable 'region' are included in the analysis. [**in**`range`] tells Stata to use only observations within the `range`.

Getting Help Within Stata

Stata provides three levels of on-line help: 1) the lookup command, 2) the help command, and 3) hands-on tutorials and data sets. The first level is recommended for users who know which statistical procedure they would like to apply but are unsure about how to implement it in Stata. It will refer the user to the second level of help activated by the command **help**. For example, the command **. help** summarize gives a short description of command **summarize**, its function and possible specifications. In addition, it will refer the user to related commands and sections of the Stata manuals. The user can get hands-on experience with Stata working with Stata's on-line tutorials and data sets. Vicki has the manuals available for loan.

10.5 Maple

Maple does **symbolic math**, which means that it can deal with **x** and **y** without having to know what **x** and **y** are. It does symbolic integration and differentiation, solving systems of equations, matrix algebra, and generally anything which involves symbol manipulation. Maple also does **numeric math** ("number crunching") to arbitrary precision, which means that you can ask it to calculate pi to 10,000 digits. Additionally, Maple has nice graphics, and can do plotting in 2 and 3 dimensions.

The current version of Maple is Maple V Release 4. Queen's has a campus-wide, all-platform site license for Maple, which means not only can you access it from almost any computer on campus, you can get a copy for home use.

To invoke maple, type maple. If you are using the X Window System, type `xmaple`. Use the command `quit` to end your Maple session.

To give you some idea of what Maple can do, we will show how to find integrals and derivatives. First, let us define a function to work with. This can be done can with the following command:

```
g := (x^3+2*x^2-x)/(x^3-x^2+x-1);
```

Note that the assignment operator is ':=', and that all Maple commands must end with a semicolon. To find the integral of this function with respect to `x`, you would type:

```
int(g,x);
```

It is always a good idea to check your answer, and with Maple it is easy. Here we take the derivative of the last expression. (The last result produced by Maple can be referred to using a double quote symbol.)

```
diff(",x);
```

The result does not look exactly like g. (Try it and see!) To verify that it is equivalent to g, subtract g from it and use `simplify`

```
simplify("-g);
```

To make a plot of g for x between 2 and 5, you would type:

```
plot(g, x=2..5);
```

The default type of plot is a character plot on a text terminal and a nice graph on an X display. If you want to put the graph into a PostScript file (called, say, `psfile`), you must specify that with a plotsetup command.

Index