COMPUTER SITE GUIDE

Department of Mathematics
The Ohio State University

June 12, 2008

Contents

Support 4

Notation 5

1 UNIX 6

1.1 What is UNIX? 6
1.1.1 What is an operating system 6
1.1.2 An overview of UNIX 6
1.1.3 History of UNIX 8
1.1.4 What is linux? 9
1.2 Accessing Our UNIX System 11
1.3 Logging In and Logging Out 12
1.3.1 Getting an account 12
1.3.2 Logging in 12
1.3.3 Passwords 12
1.3.4 Logging out 12
1.4 Accessing Departmental Information 13
1.5 The UNIX Shell 13
1.5.1 Entering Shell Commands 14
1.5.2 Aborting a shell command 14
1.5.3 Special characters in UNIX 14
1.5.4 Wildcards 14
1.5.5 Getting help 15
1.6 Working with Files and Directories 16
1.6.1 The UNIX filesystem structure 16
1.6.2 File names and directory names 17
1.6.3 Changing directories 17
1.6.4 Listing the contents of files and/or directories 18
1.6.5 "Dot" files 20
1.6.6 Symbolic links 20
1.6.7 The strange history of UNIX tools 21
1.6.8 Viewing the contents of a file 21
1.6.9 Moving and renaming files 22
1.6.10 Copying files and directories 23
1.6.11 Removing files 23
1.6.12 Backups 23
1.6.13 Creating directories 23
1.6.14 Renaming Directories 24
1.6.15 Removing directories 24
1.6.16 File and directory permissions 24
1.6.17 grep 26
1.6.18 find 27

1.6.19 Common file extensions . . . 28
1.6.20 Useful subdirectories to know about . . . 28
1.7 Redirecting Input and Output . . . 29
1.8 Printing . . . 29
1.8.1 Printing a File . . . 30
1.8.2 Querying the Printer . . . 30
1.8.3 Removing Print Jobs . . . 30
1.8.4 Printer Etiquette . . . 30
1.9 Advanced Shell Commands . . . 31
1.9.1 Background and foreground jobs . . . 31
1.9.2 Bash vs. tcsh . . . 32
1.10 Compressing, Tarring, and Transferring Files Between Computers . . . 35
1.10.1 Compressing files . . . 35
1.10.2 Tarring files . . . 35
1.10.3 Transferring files between computers . . . 36
1.11 Transferring Files Between Operating Systems . . . 37
1.11.1 The ASCII character set . . . 38
1.12 Summary of Commands . . . 38

2 Languages **41**
2.1 Compiling and Running Your Fortran, C, and C++ Programs . . . 41
2.1.1 Flags . . . 43
2.1.2 Libraries . . . 43
2.1.3 UNIX “quirks” . . . 44
2.2 Running TEX and LATEX Programs . . . 44

3 Text Editors **46**
3.1 Emacs . . . 46
3.1.1 Starting and stopping emacs . . . 48
3.1.2 Saving files and backup files . . . 49
3.1.3 Files . . . 49
3.1.4 Moving the cursor . . . 50
3.1.5 Search and replace . . . 51
3.1.6 Deletions and cut and paste . . . 53
3.1.7 Multiple windows . . . 54
3.1.8 Odds and ends . . . 55
3.1.9 Modes . . . 56
3.1.10 Compiling a program . . . 57
3.1.11 Non-printing characters . . . 58
3.2 Vi and Vim . . . 58
3.2.1 Starting and stopping vim . . . 58
3.2.2 Moving the cursor . . . 59
3.2.3 Enter edit mode . . . 60
3.2.4 Exiting edit mode . . . 60
3.2.5 Working with text . . . 60
3.2.6 Searching and replacing text . . . 60
3.2.7 Miscellaneous commands . . . 61

4 Odds and Ends **62**
4.1 Make . . . 62
4.2 Debugging Your Programs . . . 66
4.2.1 What is a debugger? . . . 66
4.2.2 Basic gdb commands . . . 67
4.2.3 Ddd, a GUI front end for gdb . . . 68

4.2.4 Basic dbx commands 70
4.2.5 Basic ldb commands 71

Support

Computer support is usually available between 8am to 5pm Monday through Friday at

Office: 430 Math Tower
Phone: 292-4962
Email: support@math.ohio-state.edu

We encourage all non-emergency questions and/or comments to be sent to us via email rather than in person or by phone. We use a system that helps us track and monitor email sent to `support@math.ohio-state.edu`. When you send email to this address, it is entered into the system and one of us is assigned to that job. In addition, it sends a response to you containing a unique "ticket" number for your inquiry. This helps us to ensure that all requests are taken care of and it allows us to easily view what needs to be done. Any follow-up emails should be sent as a reply to the response you received. In this way, any follow-up emails can be viewed by us in the same support "ticket" as the initial inquiry. (If new emails are sent on the same subject, we end up having multiple tickets open for the same issue, which makes it harder for us to track and manage what needs to be done.) Finally, when we close the "ticket", we will send you an email notifying you that the issue has been resolved. *Please do not respond to this email unless you believe that the issue is not fully resolved.*

Notation

In this document we observe the following conventions.

Notation Conventions	
<*key*>	Denotes an individual key you press. For example, <`ENTER`> denotes the "enter" or the "return" key.
`C-`*key*	Denotes a control key you press. You get this character by **holding down** the control key and pressing this key. For example, `C-b` is obtained by holding down the control key and pressing the "b" (or the "B") key.
`typewriter font`	Denotes a command you type or a UNIX tool. Also, a file name or a directory name.
italics font	Denotes information you must supply to a command.

The following conventions are used in describing UNIX commands. These agree with those commonly used in UNIX documentation.

UNIX Conventions	
`[`*construct*`]`	Square brackets around a construct indicate that it is optional.
`...`	Three dots indicate that the immediately preceding construct can be repeated one or more times.
`[...]`	This indicates that the immediately preceding construct can be repeated zero or more times.

Chapter 1

UNIX

1.1 What is UNIX?

1.1.1 What is an operating system

A computer, by itself, is simply one or more pieces of hardware which are connected together in various ways. An *operating system* is the software which brings this machine to life. There are a number of tasks that the operating system must perform. On the one hand, it must enable all the pieces of hardware to talk to each other, including the processor, central memory, disk drives, sound cards, video cards, and input-output devices. On the other hand, it must allow you to communicate with the computer to do useful work. And it must do everything in between, such as, resource allocation, scheduling, input/output, data management, and security.

There are a number of different classifications of operating systems:

- **multiuser**: Two or more users can be working on the computer at the same time.
- **multiprocessor**: The hardware contains more than one CPU (Central Processing Unit) and the operating system can keep all of them busy at the same time.
- **multitasking**: Two or more programs can run currently on one CPU. For example, you might be running one program and editing another at the same time.
- **multithreading**: Different parts of a single program can run concurrently. For example, if a number of users are running the same program, it is inefficient for each of them to have a separate copy of the program; instead, there can be one copy and each thread keeps track of what each user is doing. This is particularly useful on the web where a large number of users (possibly hundreds or thousands) can access one web site at a time.
- **real time**: The operating system responds instantly (well, almost instantly) to input. This is often needed when the computer controls a machine in a factory.

The operating systems you are probably most familiar with are Microsoft Windows (1.0, 2.0, 2.1x, 3.0, 3.1, 95, 98, ME, 2000, XP, Vista, etc.), Macintosh operating systems (OS X), and UNIX (including AT&T UNIX, Solaris, Linux, and Free BSD). In this document we are primarily concerned with UNIX. The "flavors" of UNIX we have are Linux on our PCs and BSD on our Macs.

1.1.2 An overview of UNIX

UNIX is a multiuser, multitasking operating system. That is, the operating system allows many users to run on the computer at the same time. It contains a scheduler which apportions all the tasks which are trying to use the CPU. In addition, a task can relinquish control voluntarily if it is waiting for some external event (such as user input). UNIX provides protection between tasks so that they don't "clobber" each other or the operating system itself.

A major reason for UNIX's longevity is that portability was designed into it from the beginning. UNIX is functionally organized into three parts:

1. The *kernel* is the low-level part of the operating system — the part the user never sees.
2. The *shell* is that part of the operating system which interfaces with the user.
3. The *tools* and *applications* are the programs that the user runs to interface with the computer.

The kernel

The kernel is the core of the UNIX operating system. It is a large piece of software that is loaded into memory when the computer is turned on. This is the part of the operating system which is specific to each type of computer and which has to be able to interface with all the hardware in the computer. Some of it has to be written in assembly language, but the majority is written in C. Thus it is easily ported to any computer which has a C compiler.

The shell

This is the interactive user interface with the operating system and it is the same on any computer running UNIX. The reason that UNIX is so portable is the the kernel and the shell are completely separate. The user never sees the kernel, which varies from computer to computer, but instead interacts with the shell, which is the same on all computers. Thus users can easily work on computers ranging from small minicomputers or workstations all the way up to the latest supercomputers.

There are a number of different shells in UNIX, all of which have somewhat different functionality. Some of the most common are:

- the Bourne shell, which is called *sh*,
- the Korn shell, which is called *ksh* and is a superset of *sh*,
- the Bourne again shell, which is called *bash* and is a superset of *sh*,
- the C-shell, which is called *csh*, and
- the TC-shell, which is called *tcsh* and is a superset of *csh*.

The traditional Unix shell from the very early days is the Bourne shell. Each of the other shells includes additional functionality, but all in somewhat incompatible ways. Some of this additional functionality is:

- Command/filename completion, so you only have to type part of the name of a command or file and the shell will try to complete it (which can be very helpful if you have a *looooong* file name).
- Command line editing, so you can fix errors in the command you are presently entering or reinvoke previous commands.
- History manipulation, so you can recall previous commands.
- Aliases, so you can define or redefine commands.
- Job control, so you can execute programs that will continue to run after you log out.

The tools and applications

The tools can be thought of as the low-level commands that are commonly needed to use the computer. These include logging in and out, working with files and directories, getting help, printing files, copying files between computers, logging in to remote computers, etc. (Frequently there is only one command which performs a specific task. The applications can be thought of as higher-level programs that are commonly used to perform useful work.)

These include sending and receiving e-mail, editing files, compiling and debugging programs, viewing graphics files, etc. (Frequently there are a number of different applications which can perform similar tasks.)

Odds and ends

The UNIX kernel has to be very flexible because of all the possible permutations of hardware on which it runs. In addition, there are many parameters that can be set on each computer. Thus, there needs to be a way to save all the information needed when the computer is booted. All this information is stored in system configuration files, which are read by the kernel at boot time. These files are stored in easily accessible subdirectories and are normally written in plain text for easy modification by system administrators.

This accessibility is also used at the user level. Many aspects of the behavior of the shell can be modified by the user. These files are normally stored in the user's main directory and are called *dot files* because the first character in the file name is a dot (i.e., a ".").

1.1.3 History of UNIX

Note: The rest of this section is of historical interest and can be skipped, if desired.

In the early 1960's there was a multi-organizational effort at Bell Labs, the research arm of AT&T (American Telephone and Telegraph), to develop a dependable timesharing operating system. It is important to understand the state of computer systems in the 1960's (and even into the 1970's and 1980's). Computer systems from different companies could not talk to each other; frequently, even different computer lines made by the same company could talk to each other only with *great* effort. In addition, operating systems were usually designed for a single computer. Frequently if a business upgraded to a larger more powerful computer, the operating system would change. This usually meant that all the company's data had to be transferred to the new computer at a *huge* expense in time and money. Operating systems were significantly different from each other (even in the same company) and it took a significant effort to learn a new one. Frequently, there were only a small number of people at a site who understood how to do anything but the most basic operations. If they were unavailable, well you just waited until they returned (even if they were on vacation).

Although Bell Labs' effort failed, a few of the survivors continued working sporadically on developing a "friendly", interactive operating system. Their success followed from their distinctive approach to software design: solve a problem by developing and connecting simple tools, rather than by creating a large monolithic monstrosity. In the early 1970's UNIX was born, but the software had to be written in assembly language. The next fundamental advance was their development of the computer language C so that UNIX could be written in a high-level language. Their philosophy was: "Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface."

Note: Why "C"? There already was APL (A Programming Language) and BCPL (Basic Combined Programming Language) and B, which was developed at Bell Labs and was a "stripped down" version of BCPL. So, why not "C"?

Around 1975 AT&T began licensing UNIX, including the source code, to the government, private industry, and universities. The University of California at Berkeley in particular made their own enhancements to UNIX (called BSD — the Berkeley Software Distribution) and spread the gospel through academia. Slowly vendors started adding UNIX as an optional operating system on their computers, and

universities were often at the forefront of this change. No longer were users at the mercy of their organization's central computer group — if they could find the money, they could buy the computer that best suited their needs. They were willing to put up with this *strange* new operating system because of the freedom it gave them. It is hard to understand today how much of an impediment "centralized computing" was — the fall of central computer groups was as much of a revolution as the fall of Communism. Businesses also liked UNIX because programs could be written in C and run on *all* a company's computers; in addition, data could be easily moved between all these computers. Their pressure forced vendors to continue offering and enhancing UNIX, although each one usually had a somewhat different version of UNIX. (It even became available on Cray supercomputers in the early 1980's under the name UNICOS.)

During this time many vendors were not happy, because the last thing they wanted was for users to be able to choose their own computers. Vendors were very happy dealing with central computer groups — it made their selling job much easier because these groups would then force their users to take whatever computer was offered. UNIX weakened these groups so they wanted to splinter UNIX. AT&T entered into an alliance with Sun Microsystems to create the "one true UNIX standard". In response many other vendors, led by IBM, formed a special interest group, the Open Systems Foundation (OSF), to lobby for an "open" UNIX within the UNIX community. (The joke at the time was that the only thing *open* about the Open Systems Foundation was its name.) Today, there are a number of UNIX-based systems: Solaris from Sun Microsystems, HP-UX from Hewlett-Packard, AIX from IBM, and Tru64 UNIX from Compaq. In addition there are many freely available UNIX and UNIX-compatible implementations, such as FreeBSD and NetBSD. And, of course, there is Linux.

1.1.4 What is linux?

Linux is a particular variant of UNIX. Its history actually began in 1983 with the initial announcement of the GNU (GNU's Not Unix) Project. In 1971 when Richard Stallman started his career at MIT he worked in a group which used *free software* exclusively. By "free software" they did not mean that it did not cost any money to acquire. Instead, they meant for the users of the software:

0. The freedom to run the program, for any purpose.
1. The freedom to study how the program works, and adapt it to your needs. Access to the source code is a precondition for this.
2. The freedom to redistribute copies so you can help your neighbor.
3. The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. Access to the source code is a precondition for this.

By the 1980's, however, almost all software was *proprietary*. That is, it cost money to acquire, usually you only obtain an executable code (i.e., you could not access the source code), and you were restricted in how you could use it.

The GNU Project, with Stallman as its leader, was designed to bring back the cooperative spirit in computing by removing the obstacles to cooperation imposed by the owners of proprietary software. Since every computer user needs an operating system, they began by making an operating system which was compatible with UNIX. They began by writing their own C compiler (called *gcc*). By the early 1990's they had written everything except the kernel.

> It was 1991, and the ruthless agonies of the cold war was gradually coming to an end. There was an air of peace and tranquility that prevailed in the horizon. In the field of computing, a great future seemed to be in the offing, as powerful hardware pushed the limits of the computers beyond what anyone expected.
>
> But still, something was missing.
>
> And it was the none other than the Operating Systems, where a great void seemed to have appeared.
>
> For one thing, DOS was still reigning supreme in its vast empire of personal computers. Bought by Bill Gates from a Seattle hacker for $50,000, the bare bones operating system had

> sneaked into every corner of the world by virtue of a clever marketing strategy. PC users had no other choice. Apple Macs were better, but with astronomical prices that nobody could afford, they remained a horizon away from the eager millions.
>
> The other dedicated camp of computing was the Unix world. But Unix itself was far more expensive. In quest of big money, the Unix vendors priced it high enough to ensure small PC users stayed away from it. The source code of Unix, once taught in universities courtesy of Bell Labs, was now cautiously guarded and not published publicly. To add to the frustration of PC users worldwide, the big players in the software market failed to provide an efficient solution to this problem. (Ragib Hasan, "History of Linux", http://netfiles.uiuc.edu/rhasan/linux)

There was a "minimal" operating system, called MINIX, written by Andrew Tanenbaum, a Dutch professor. It was suitable for teaching students the inner workings of a real operating system and was written for the Intel 8086 microprocessor. It was not the solution.

In 1991 Linus Torvalds, a second year student of Computer Science at the University of Helsinki, decided to write a better kernel for his own computer:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki


Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones.  This has been brewing
since april, and is starting to get ready.  I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to
work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want.  Any suggestions
are welcome, but I won't promise I'll implement them :-)

              Linus (torvalds@kruuna.helsinki.fi)

PS.  Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never
will support anything other than AT-harddisks, as that's all I have :-(.
```

The first few versions appeared by the end of the year. Soon more than a hundred people were helping write, modify, and debug the software. Then more than a thousand, then more than ten thousand, etc. Powered by the tools and applications from the GNU project, it was licensed under the GNU General Public License, thus ensuring that the source codes would be free for all to copy, study, and to change.

By 1993 there were 100,000 Linux users and by 1995 more than 1,500,000. In 1995 Red Hat Software, Inc., was founded. (Linux itself is free, but vendors are also free to put the operating system, as well as lots of useful software packages, on CD's, and include instructions on how to load everything on computers.)

For more, and more current, information on Linux see the articles in Wikipedia and by Ragib Hasan (reference above).

1.2 Accessing Our UNIX System

If you are outside the math department's network, you access our system by logging onto `math.ohio-state.edu` using *ssh*. If you are using a GUI (i.e., a Graphical User Interface) to run ssh, simply follow the instructions. If you are using a terminal, enter

```
ssh -Y username@math.ohio-state.edu
```

(to be sure that X-windows "tunnels" through the connection) or

```
ssh username math.ohio-state.edu
```

(or even

```
ssh math.ohio-state.edu
```

if your username on the remote machine is the same as on your local machine. (The machine you were originally on is called your *local* machine, and the machine you have connected to is called the *remote* machine.)

Developed by SSH Communications Security Ltd., ssh (short for "secure shell") is a program which logs you *securely* into another computer over a network. Its advantages are:

- Your entire login session, including your username and password, is encrypted so that a hacker cannot steal your password
- It can connect most computers, including those running Linux, Macintosh, UNIX, and Microsoft Windows operating systems.
- It supports X-windows, the window system in UNIX.

If you are inside the math department's network, you can log onto any machine inside our network that you have access to. Use the above command but replace `math.ohio-state.edu` by the machine name. That is, you do not need to include `.ohio-state.edu`.

Dumb terminals

Computer terminals are referred to as *dumb* terminals if they have only the minimum amount of power required to send characters to the UNIX system and receive characters from UNIX. Personal computers are often used to emulate dumb terminals when they are connected to a UNIX system. When you are using your local computer as a dumb terminal, you cannot access X-windows or any graphical images.

Smart terminals

Smart terminals interact with the UNIX system at a higher level because the interaction can include icons, windows, menus, and mouse actions.

If your local machine is running UNIX, then it is automatically a smart terminal. Macintosh computers running OSX are also smart terminals if you fire up X-windows on you local machine. Microsoft Windows computers are normally dumb terminals, but can be made smart if you use one of the following software packages:

- `XWIN32` is a commercial product but OSU has a site license. It is very easy to install and it works well.
- `cygwin` is a UNIX environment for Microsoft Windows. It contains GNU's tool and applications and so allows you to use UNIX commands instead of using Microsoft Windows. It contains `ssh` and X-windows and so allows you to connect directly with UNIX computers. It is free, but it is somewhat difficult to install.
- A number of linux distributions have *live* DVDs which allow you to boot directly from the DVD and to store files you create on the hard drive. This enables you to use linux directly on a PC without having to go to the trouble of actually installing linux on the hard drive — which is something that a novice should not be doing.

1.3 Logging In and Logging Out

To ensure security and organization on a system with many users, UNIX machines employ a system of user accounts. The user accounting features of UNIX provide a basis for analysis and control of system resources, preventing any user from taking up more than his or her share, and preventing unauthorized people from accessing the system.

1.3.1 Getting an account

If you do not already have an account on our system, come to room MW 430 and fill out an account form. You must personally give the application to one of our computer staff members in MW 430 and provide a picture ID for identification. They will assign you an initial password. *You should then log in to your new account and change this password.* We discuss this below.

1.3.2 Logging in

When you sit down to log in to one of our computers, you will see a prompt that says

```
login:
```

asking for your user name. Type in your user name and hit return. Next you will see the prompt

```
password:
```

Enter your password and hit return. For security reasons your password will not be displayed, so be sure to type carefully.

1.3.3 Passwords

If this is the first time you have logged in to our system, you should change your password by typing

```
passwd
```

You will need to enter your *current* password (to prevent someone else from sneaking up and changing your password), followed by your desired new password *twice*. Your password cannot be based on any words and must be at least 6 characters long. Here are some rules for selecting a good password:

- **Do not** use any part of your name, your spouse's name, your child's name, your pet's name, or anybody's name. Do not use any backward spellings of any name, either.
- **Do not** use an easily-guessable number, like your phone number, your social security number, your address, license plate number, etc.
- **Do not** use any word that can be found in an English or foreign-language dictionary.
- **Do not** use all the same letter, or a simple sequence of keys on the keyboard, like `qwerty`.
- **Do** use a mix of upper-case and lower-case letters, numbers, and control characters.
- **Do** use at least six characters.

If the system does not accept the password you type in, you will be asked to try again with a password which is more difficult to guess.

1.3.4 Logging out

When you're ready to quit, type the command

```
exit
```

Before you leave, make sure that the login prompt appears, indicating that you have successfully logged out. If you have any suspended processes or processes running in the background (see section 1.9.1), the UNIX system will require you to handle them before it will let you log out.

1.4 Accessing Departmental Information

There is a huge amount of information about the department in our departmental web page

`http://www.math.ohio-state.edu`

There are a number of browsers you can use to access the web. One that will work on all of our UNIX machines is mozilla. Simply type

```
mozilla &
```

and go! (The reason for using "`&`" is discussed in section 1.9. Briefly, it puts the command in the background so that you can use the terminal for other work and also have mozilla running at the same time.)

You can also access some computer information on our web site. Simply click on `Computer` at the top of the page or the bottom, or click on "• `Computer Support`" near the bottom of the page. We know that this web page could use some improvement. (As our head system administrator has said more than once: "It sucks, but it's better than nothing".) And, yes, we will improve it over time.

One piece of information that people often need is the departmental phone list — which includes the person's office and username as well as the phone number — for faculty, staff, and graduate students. You can easily access this through our web page. However, there is a local command which is much faster. Simply type

```
phone
```

and you are in the text oriented web browser `links`. Some simple commands to move you around in it are:

- `<SPACE>` move forward one screen
- `b` move backward one screen
- `/` search for text (enter the text in the window, move the mouse over "`[ OK ]`", and click on it)
- `n` continue the present search
- `q` quit (move the mouse over the "`[ YES ]`" and click on it)
- `C-c` quit (move the mouse over the "`[ YES ]`" and click on it)

1.5 The UNIX Shell

The shell is your interface with the UNIX system — the middleman between you and the kernel. It accepts a command, interprets the command, executes the command, and then waits for another command. The shell displays a *prompt* to notify you that it is ready to accept your command. This prompt will probably look something like

[*username*@*hostname directoryname*]$

The *hostname* gives the name of the machine you are presently running on, and the *directoryname* shows the directory you are presently in. For example,

```
[zola@math p_1]$
```

denotes the user "`zola`", running on the computer called "`math`", and presently in the directory "`p_1`" (as seen in Figure 1.1).

Note that the shell is itself a running program. A formal definition of a *process* is that it is a single program running in its own address space. When you execute a non-built-in shell command, the shell asks the kernel to create a new subprocess (called a *child* process) to perform the command. The child process

exists just long enough to execute the command. The shell waits until the child process finishes before it will accept the next command. A single CPU computer can only run one process at a time, but by switching between the processes it can appear that many processes are running at the same time. Even if you are the only user on a computer, it is not uncommon for there to be over 50 processes running.

1.5.1 Entering Shell Commands

The shell recognizes a limited set of commands, and you must give commands to the shell in a way that it understands. Each shell command consists of a command name, followed by command options (if any are desired) and command arguments (if any are desired). The command name, options, and arguments, are separated by one or more blank spaces, i.e.,

command_name [-*option* ...] [*argument* ...]

Note: "[ ]" signifies optional parts of the command that may be omitted.

The command name is the name of the program you want the shell to execute. The command options (if used) usually begin with a dash (i.e., "-") and allow you to alter the behavior of the command. The arguments (if used) are the names of files, directories, or programs that the command needs to access.

Notation: Unlike DOS, the UNIX shell is case-sensitive, meaning that an uppercase letter is not equivalent to the same lower case letter (e.g., "A" is not equal to "a"). Almost all UNIX commands are in lower case.

1.5.2 Aborting a shell command

If you need to abort the current command, enter `C-c`.

1.5.3 Special characters in UNIX

UNIX recognizes certain special characters as command directives. If you use one of the UNIX special characters in a command, make sure you understand what it does. The special characters are:

/ < > ! $ % [ ] ^ & * | { } ~ and ;

We will discuss the meaning of many of these characters, and how to use them, in this document.

1.5.4 Wildcards

Wildcards are used to perform a command on multiple files and/or directories. For example, suppose you want to list all the files in a directory that begin with the letter "a", or suppose you want to list all files that are two letters long and begin with the letter "a". The two most common wildcards are

- `*` match any string of characters (from 0 characters to an infinite number of characters)
- `?` match any single character

The "*" wildcard matches anything. For example, the command

```
ls a*
```

lists all files that begin with the letter "a", including the file named "a" (if it exists).

The "?" wildcard matches *exactly* one character. For example, the command

```
ls a?
```

lists all files that begin with the letter "a" and are exactly two characters long.

To be more creative, the command

```
ls a?.*
```

lists all files that begin with the letter "a", then a single character, then a dot, then 0 or more characters.

Note: The above discussion is not *quite* correct, and so we will fill in the details here. However, you need to understand file and directory pathnames and "dot" files, which are all discussed in the next section. A "/" in a *path name* can only be matched by an **explicit** "/" in the pattern, not by a "*" or a "?". Furthermore, if a name either begins with a "." or has a "." which immediately follows a "/", then that "." must be matched by an **explicit** ".", not by a "*" or a "?".

Another wildcard that is quite useful is the construct "`[ ]`" which is more specialized than the wildcard "?". Whereas "?" represents *any* character (e.g., a digit or a lower-case letter or an upper-case letter or a punctuation mark, etc.), "`[ ]`" allows you to restrict the characters. Rather than trying to explain it in detail, we will only show a few useful examples.

The wildcard "`[`*characters*`]`" represents any individual character in *characters*. For example,

`[aeiou]` represents the five vowels "a", "e", "i", "o", or "u".

`[sS]` represents either the lower case "s" or the upper case "S".

`[13579]` represents the odd integers.

It would be quite annoying if you had to represent any lower case letter by `[abcdefghijklmnopqrstuvwxyz]`. Instead, you use "`-`" to denote a range of characters. For example,

`[a-z]` represents any lower-case letter,

`[C-M]` represents any upper-case letter between "C" and "M" inclusive,

`[0-9]` represents any number,

`[1-7]` represents any number between 1 and 7 inclusive, and

`[a-zA-Z]` represents any letter ("`a-z`" represents any lower case letter, `A-Z` represents any upper case letter, and putting them together means "`a-z`" or "`A-Z`").

1.5.5 Getting help

To access the on-line manuals, use `man` (short for "manual"), followed by the name of the command you need help with. For example,

```
man ls
```

returns a lengthy discussion of the `ls` command. To get help on using the `man` command itself, enter

```
man man
```

Sometimes you cannot "quite" remember the name of the command you want to use — which is not surprising since command names in UNIX are often rather "quirky". You can search through a database for keywords which will (hopefully) lead you to the command you want. Just type

`apropos` *keyword ...*

or

`man -k` *keyword ...*

and look through the responses. These responses will normally be a command name followed by a brief description. (Between these two you will see the page number and, possibly, the section number where the "man" file is stored — which you are normally not interested in.)

As is common in UNIX, this command is frequently not as useful as it might seem. Sometimes the command you are searching for does not have a "man" file, so its keywords were not included in the database, and so it will not show up in the responses. Alternatively, you might get *hundreds* of responses and find it is too much trouble to read through them all. It is also possible to search the web for the command. Using google with the keywords "UNIX manual *keyword* ..." will frequently give you exactly what you are looking for.

Note: For example, if you forget the command for renaming a file, you can use the keywords "UNIX manual rename", but this will discuss the command `rename` (which is probably not what you want. However, the keywords "UNIX manual rename files" will lead you immediately to the command `mv` (which is probably what you want). The keywords "UNIX tutorial rename" will lead you to a more general discussion of how to rename files. In there you will find the command you are searching for.

In addition, many programs respond to help arguments by displaying a brief list of options and a quick description. Try typing the following:

command `--help`
command `-h`
command `-help`

Hopefully, at least one of these will work — but there are no guarantees! (Remember, you can always use google to find a discussion of *command*.)

1.6 Working with Files and Directories

1.6.1 The UNIX filesystem structure

All the information stored in a UNIX computer is kept in a filesystem. Whenever you interact with the UNIX shell, it considers you to be located somewhere within this filesystem. Your present location in the filesystem is called the *current working directory*.

The UNIX filesystem is hierarchical, i.e., it resembles a tree structure. The tree is anchored at its base, which is called the root and designated by a slash "/". Every item in the UNIX filesystem tree is either a file or a directory (which you can think of as a file folder). A directory can contain files, and it can also contain other directories. If directory `ch_1` is contained within directory `thesis`, then `ch_1` is called the *child* of `thesis` (or a subdirectory of `thesis`) and `thesis` is called the *parent* of `ch_1`. A directory in the filesystem tree may have many children — but it can only have one parent. A file can hold information, but it cannot contain other files or directories.

To describe a specific location in the filesystem hierarchy, you must specify a *path*. The path to a location can be defined as an absolute path from the root directory, or as a relative path, starting from the current working directory. When specifying a path, you simply trace a route through the filesystem tree, listing the sequence of directories you pass through as you go from one point to another. Each directory listed in the sequence is separated by a slash. To explain this in more detail we show a specific example of part of a filesystem tree in Figure 1.1. We will use this specific example throughout this section.

Note that the root directory's name is simply "/", and "/" is also used to separate the directories on a branch of this tree. It is very easy to tell which is which. If "/" has no name before it, then it stands for the root directory. For example, `/home/zola/Hmwk1/p_2` refers to the subdirectory `p_2`, which is at the lower right corner of the tree. If "/" has a name before it, then it refers to a path which begins at your current working directory. For example, if you are presently in the subdirectory `home`, then `zola/Hmwk1/p_2` also refers to `/home/zola/Hmwk1/p_2`.

UNIX provides the shorthand notation of "`.`" to refer to the current directory, and "`..`" to refer to the parent directory. For example, if you are presently in the subdirectory `Thesis`, you can refer to the directory `p_2` by `../Hmwk1/p_2` as well as by `/home/zola/Hmwk1/p_2`.

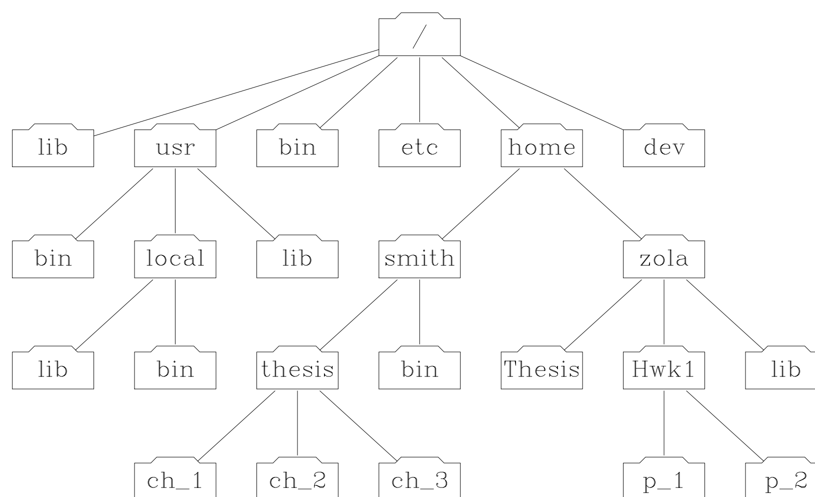


Figure 1.1: Part of the filesystem tree

In UNIX you have your own directory, called your *home* directory. You can always refer to this directory by entering "`~`". For example, if you are the user `smith` you can refer to your subdirectory `ch_1` by `~/thesis/ch_1` or by `/home/smith/thesis/ch_1` or even `~smith/thesis/ch_1` . Note the last path name: `~smith` is always the home directory of the user `smith`. If you are **not** `smith`, then `~smith` refers to smith's home directory. If you are `smith`, then "`~`" or `~smith` works equally well.

It is very important to understand that your home directory does not provide privacy!!! Normally, other users can go into your home directory and read or copy any of your files (well most all of your files anyway). They cannot change or delete your files — but they can look at them or copy them. In section 1.6.16 we will discuss how you can easily change this behavior.

Note: Our computer support staff can read and manipulate *all* the files in the filesystem. If you don't trust anyone, keep all your files at home on your own computer — and don't connect it to the web.

1.6.2 File names and directory names

When creating files and directories on UNIX, it is safest to only use the characters A-Z, a-z, 0-9, the period (i.e., "`.`"), and the underscore (i.e., "_") characters. (Unlike DOS, the UNIX shell is case-sensitive, meaning that an uppercase letter is not equivalent to the same lower case letter so that, for example, "A" is not equal to "a".)

File and directory names can be any length (well, any *reasonable* length). You should have a very good reason before using more than 20 or 30 characters. File extensions (see section 1.6.19) can also be any length, but there are no common file extensions which are more than 5 characters in length.

1.6.3 Changing directories

When you log in to a UNIX computer, you are automatically placed in your home directory. To see where you are, type

```
pwd
```

(short for "print working (i.e., current) directory"). To change directories, use

```
cd
```

For example, to change to the directory `/home/zola/Thesis` use

```
cd /home/zola/Thesis
```

Notation: Although we have differentiated files from directories, *technically* they are the same. For example, a common definition of a directory is that it is a file which contains links to other files. To UNIX a directory is just another file, but a file which contains the addresses of other files. UNIX can keep track of which are which, and you will see how in subsection 1.6.4. Some UNIX tools only work with "ordinary" files, while others only work with directories, while others work with both. To differentiate these three possibilities we will introduce a new word:
For our purposes, we want to keep files *and* directories *separate, so we will use the name "*path*" when we want to refer to either. In addition, we combine* file names *and* directory names *into "*pathnames*".*

1.6.4 Listing the contents of files and/or directories

The `ls` (short for "list") command allows you to see the contents of a directory, and to view basic information (like size, ownership, and permissions) about files and directories. Since it has numerous options, see the manual page on `ls` (type "`man ls`") for a complete listing. The `ls` command also accepts one or more arguments, which can be directories and/or files (i.e., paths).

For example, type

```
ls .
```

or

```
ls
```

(which is equivalent) to see all the pathnames (i.e., names of files and directories) in your current working directory. Recall that "`.`" is the name of your current directory; when you list a directory, it actually lists all the pathnames in that directory.

You can type

```
ls *.c
```

to list all the files whose extension is "c". (This will also find any directories which end in "`.c`", but it is generally considered a bad idea to have extensions on directories — and so we will ignore this possibility.) Or type

```
ls a?.c
```

to list all files whose name is two characters long, with the first being an "a", and whose extension is "c". As another example, if you have the subdirectory `bin` in your home directory, type

```
ls ~/bin
```

or

```
ls ~/bin/
```

to list all the pathnames in this subdirectory.

Actually, we have not been *quite* exact in this discussion because the above commands need not give you all of the paths in your directory: they do not include any "dot" files, i.e., files whose first character is a "`.`". (You probably even have a few directories whose names begin with a "`.`". However the phrase "'dot' files", even if the file is a directory, has been around for so long that it is the common terminology.) See subsection 1.6.5 for more information on "dot" files.

To list *all* the paths, you have to include the option "`-a`". For example,

```
ls -a
```

lists *all* the paths (i.e., including the "dot" files) in your current directory. If you type

```
ls -F
```

you can determine which are which:

- if a pathname ends with a "/", it is a directory; otherwise, it is not.
- if a pathname ends with a "*", it is an executable file (more about this later).
- if a pathname ends with a "@", it is a link to a file (see subsection 1.6.6).
- if a pathname does not end with any of the above special characters, it is a file.

Alternatively, you can try to use colors to differentiate pathnames by typing

```
ls --color=tty
```

There is an important difference between

```
ls .
```

and

```
ls *
```

if your current directory contains subdirectories. The reason is that "`ls` *directoryname*" lists all the files in the given directory. Thus "`ls .`" lists all the files in your current directory. However, "*" expands to all the non-dot files in your directory; if these files include directory files, then the contents of these directories will be listed as well as the name of the subdirectory itself. For example,

```
ls a*
```

will list all the files in your current directory which begin with the letter "a"; if any of these are directory files, then the contents of these subdirectories will also be listed. Frequently, this is not what is desired. If not, use the "`-d`" flag so that only the directories will be listed and not their contents. For example,

```
ls -d a*
```

will list just the files in your current directory which begin with the letter "a"; even if any of these are directory files, the only directory name will be printed out and not the contents of the directory.

Interpreting the long format

The `ls` command produces a "long" listing, i.e., more information, if you specify the "`-l`" option. For example, if you are in the directory `~zola/Hwk1` and you type

```
ls -l
```

you might see

```
-rwxr-xr-x    1 zola  guest    13568 Jan 16 11:48 check
-rw-r--r--    1 zola  guest      123 Jan 16 11:47 check.c
-rw-r--r--    1 zola  guest      972 Jan 16 11:48 check.o
-rw-r--r--    1 zola  guest    28149 Jan 16 11:47 hmwk-1.tex
-rw-r--r--    1 zola  guest    17214 Jan 16 11:50 hmwk-2.tex
drwxr-x---    2 zola  guest     4096 Jan 16 11:53 p_1
drwx------    2 zola  guest     4096 Jan 16 11:52 p_2
lrwxrwxrwx    1 zola  guest       12 Jan 16 11:54 scan.dat -> p_1/scan.dat
```

We will discuss what this all means shortly, but first we concentrate on the last item (i.e., the last column), which gives the pathname. The second pathname is a C language source file; the third is its object file and the first is its executable file (denoted by the "`x`" in the fourth character of the first item (see 1.6.16)). Next, we have two TEX files. The sixth and seventh pathnames are the subdirectories of this directory (denoted by the "`d`" in the first character of the first item (see 1.6.16)). And, finally, the eighth file is actually contained in the directory `/home/zola/Hwk1/p_1` , but it can be accessed as if it was in this directory (i.e., `/home/zola/Hwk1`). We discuss this in detail in subsection 1.6.6.

Now we interpret the entire listing:

- The first item determines the type of path and its protections. The first character determines the type of path:

  `-` is a file

  `d` is a directory and

  `l` is a symbolic link to a file or directory (see subsection 1.6.6).

  (There are several other possibilities, but they are more advanced and we will not discuss them.) The remaining nine characters determine the protection, which we discuss in section 1.6.16.

- The next item shows how many links this file has (we won't worry about this).
- The next two items give the user's name and group; the group indicates the type of user (student, faculty, staff, etc.).
- The next item gives the size of the file in bytes.
- The next three items show the date and time when the file was last modified.
- The last item gives the pathname. A "`/`" after the pathname indicates it is a directory (recall that the "`d`" in column 1 does also); a "`*`" indicates it is an executable file; and a "`->`" indicates that this pathname is a symbolic link to another pathname.

  You can do a long listing which also includes all the "dot" files by typing

```
ls -al
```

Continuing the example shown in Figure 1.1, this will list all of the above files as well as

```
drwxr-xr-x    4 zola  guest     4096 Jan 16 11:54 .
drwxr-xr-x    5 zola  guest     4096 Jan 16 11:46 ..
```

(Recall that we are presently in the directory `/Tilde zola/Hwk1` .) The first pathname is "`.`", which is the directory `/home/zola/Hwk1` (denoted by the "`d`"). The second pathname is "`..`", which is the parent directory `/home/zola` .

1.6.5 "Dot" files

We stated previously that the kernel saves boot-time parameters in files called system configuration files. Parameters can also be set for the user. Many aspects of the behavior of the shell can be modified by the user. These parameters are usually stored in the home directory. However, they would clutter up a listing and so they are hidden by making them *"dot" files*, that is, files whose first character is a "`.`". Most of these files can be modified directly by the user.

In addition, many software applications also set parameters in "dot" files or create "dot" subdirectories into which they store all necessary information. For example, there is probably a directory `.netscape` or `.mozilla` in your home directory.

1.6.6 Symbolic links

Suppose you are working in a particular directory and you frequently need to refer to a file in another directory. For example, you might have one data file which is shared between many programs in many different directories. It can get quite annoying to have to type the complete pathname of this data file — and you might even forget where that <*insert your favorite adjective*> file is. You might be tempted to copy the data file to each directory where it is needed. **However, this is a very bad idea for at least two reasons: first, you are wasting disk space; and, second and much more important, you might later decide to change the data file and you would have to search throughout your entire tree branch to find, and replace, all the copies.**

UNIX allows you to refer to a file (or directory) indirectly, that is, as if it was in your current directory. By typing

`ln -s` *real_pathname* `.`

you are putting a symbolic link in your current directory to the file which is in another directory. The filename in the current directory is the same as the actual filename and you can behave as if the file is really in your current directory. However, it is not, and your local directory entry points to the actual file; if you modify the actual file, your link will point to the modified file. Note that the "`.`" means to use the same filename in the local directory. You can use a different name by typing

`ln -s` *real_pathname* *local_filename*

1.6.7 The strange history of UNIX tools

In section 1.1.3 we discussed the history of UNIX. Many of the standard tools of UNIX (such as `ls`) date back to the 1970's and were written by various people. There was no committee to standardize the interface with the users, or even to choose "good" names for these tools. (For example, the tool **`awk`** is a pattern scanning and processing language. Its name comes from its authors: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan.) Users frequently wonder why the options that work for one tool do not work for another, or why the form of the arguments for one tool are different from that of another. In addition, the various vendors that put UNIX on their computers often tried to "standardize" the tools — but they each standardized them differently! There is no way around it — you simply have to memorize what works for what tool (or use the man command).

For example, two options which we have already discussed for the `ls` command are "`-a`" and "`-l`". Some tools, such as `ls`, allow you to combine these two options as "`-al`". On the other hand, some commands require that all options be entered separately, e.g., "`ls -a -l`".

GNU has tried to bring some order out of this chaos and so has its own standard. Since GNU tools are freely available, they might eventually become the default standard. For example, the command "`ls -F`" works (we believe) on all flavors of UNIX. However the option "`--color=tty`" is a GNU addition. Thus, it is the default in Linux.

1.6.8 Viewing the contents of a file

There are several commands that will display the contents of a file on the screen. Probably the most common commands are the `more` and `less` commands (no pun intended). `More` used to be a very basic program to display one page of a file at a time, and `less` was created to add more features to this. However, `more` was modified to include many of these same features. The upshot is that they more or less do the same thing (pun intended). The command

`less` *filename*

displays the first screen of the file. Some of the basic commands are:

- <`ENTER`> to display the next line,
- <`SPACE`> to display the next screen,
- "`b`" to display the previous screen,
- "*/string*" searches forward from the current position for *string*,
- "<`ENTER`>" or just "`n`" continues searching forward for the same string,
- "*?string*" searches backward from the current position for *string*, and
- "`?`<`ENTER`>" or again just '`n`" continues searching backward for the same string.

If you would just like to display the first few lines of a file, use the command

`head` *filename*

If you would like to display a specific number of lines, add the option "*-number*" where *number* is the number of lines you want to see. To display the last few lines of a file, use the command

`tail` *filename*

(Again, "*-number*" is an option.)

For short files, the `more` or `less` commands (no pun intended) can be overkill. Instead,

`cat` *filename* ...

(short for "concatenate") displays the entire contents of the file (or files) on the screen. This only works for short files because the entire contents of the file will appear on the screen — and so it had better actually fit on the screen. (Another use for this command is shown in section 1.7.)

1.6.9 Moving and renaming files

To move or to rename an existing file, use the command

`mv` *existing_filename new_filename*

(short for "move"). You might want to rename a file, or you might want to move a file to a different directory and, possibly, rename it (in which case you have to give the complete path in *new_filename*).

One danger with this command is that if you are moving a file to a different directory and there is already a file with the same name in that directory, it is overwritten — without any indication that you have "clobbered" the file. This might be what you want to do — or it might not. For example, it occasionally happens that your fingers are in high gear but your brain is in neutral and *new_filename* is an already existing file (because your fingers have been typing a particular sequence of characters for too long). If you overwrite an important file, ... (well, you can fill in the rest).

The solution is to type

`mv -i` *existing_filename new_filename*

The "`i`" option stands for "interactive" — if a file would be overwritten, you are asked if that is what you really want to do. If it is, you reply "`y`" for yes; otherwise, it is not overwritten. Since this is so important, `mv` has been aliased to "`mv -i`" for just this eventuality. This is discussed in subsection 1.9.2.

To move a file to a new directory without changing the filename, type

`mv` *filename directoryname*

or

`mv` *filename directoryname/*

The latter is preferable because if you make a typing mistake and *directoryname* does not exist, the former will copy the file *filename* to the **file** *directoryname*. This is not a fatal mistake since you have not deleted the file. However, you may have great difficulty in figuring out where it disappeared to. On the other hand, the latter command will give you the error message that the directory does not exist and then abort.

For example, you can move the file `thesis.tex` to the subdirectory `Thesis` by

```
mv thesis.tex Thesis/
```

If, instead, you want to move it to the Thesis directory, but to change its name to `contents.tex`, you would enter

```
mv thesis.tex Thesis/contents.tex
```

You can also move multiple files to a directory by typing

`mv` *filename ... directoryname*

or

`mv` *filename ...* *directoryname/*

For example, if you have the files `ch_1.tex`, `ch_2.tex`, ..., `ch_10.tex` in your current working directory, you can move them all to the subdirectory `Thesis` by

```
mv ch*.tex Thesis/
```

If you are moving files from another directory to the current working directory and you do not want to change the filenames you can use "`.`" for the current directory. For example, if you have just moved all the files `ch_1.tex`, `ch_2.tex`, ..., `ch_10.tex` from your current working directory to the subdirectory `Thesis` by the above command, you can move them all back again by

```
mv Thesis/ch*.tex .
```

1.6.10 Copying files and directories

To copy an existing file to a new file, use the command

`cp` *existing_filename new_filename*

All the examples above which use the `mv` command apply equally well to the `cp` command.

1.6.11 Removing files

To remove one or more files, use the command

`rm` *filename ...*

Again, `rm` has been aliased to "`rm -i`". However, here it has a different meaning. This option stands for *interactive mode*, meaning that you will be asked for confirmation before anything is removed. In the above example, you will be asked if you really want to remove the file *filename*, to which you respond either "`y`" or "`n`". This again prevents problems if you have mistyped *filename*.

Another useful option for `rm` is "`-f`", which forces the file to be removed without asking for confirmation. This is useful if `rm` has been aliased to "`rm -i`" (as has been done in our department).

Warning: **Make sure you really, really know what you are doing before using the "`-f`" option.**

1.6.12 Backups

The Math department does backups several times per week of *all* the files in the filesystem. However, we cannot restore any changes in files that were make since the previous backup.

1.6.13 Creating directories

To create one or more directories, use the command

`mkdir` *directoryname ...*

As with any other UNIX command, the directory name can be a full path to the directory you wish to create (like `/home/zola/Thesis`) or it can be just a directory to create in the current directory (such as `Thesis` if you are presently in the directory `~zola`).

If you want to create two or more levels of subdirectories, you have to be more careful. For example, suppose the subdirectory `ch_1` already exists and you want to create the subdirectories `ch_1/figures/eps` and `ch_1/figures/pdf` . If the subdirectory `ch_1/figures` does not already exist, you have to create it first by, for example,

```
mkdir figures figures/eps figures/pdf
```

Note that the subdirectory `figures` is created *before* its subdirectories are created. Alternatively, if you are using the GNU mkdir command you can just type

```
mkdir -p figures/eps figures/pdf
```

The option "`-p`" creates any parent directories which do not already exist.

1.6.14 Renaming Directories

As with files, you rename directories by moving them to a new name. The command

`mv` *old_directoryname new_directoryname*

renames *old_directoryname* to *new_directoryname*; all the files and subdirectories which were originally in the directory *old_directoryname* are now in the directory *new_directoryname*.

1.6.15 Removing directories

To remove an *empty* directory, use the command

`rmdir` *directoryname ...*

This will only work if the directory is empty — otherwise, it will return an error message and abort.

Another useful option for `rm` is the recursive option. The command

`rm -r` *directoryname ...*

will recursively delete a directory. This means that it will delete the directory and *all* subdirectories on this branch of the tree and *all* the files on this branch. The command

`rm -rf` *directoryname ...*

will also recursively delete the directory, but without asking for confirmation.

Warning: **Of course, you will only do this after careful thought, WON'T YOU? There once was a system administrator (well, actually, he was a faculty member — to remain nameless — who was made the administrator because, well, nobody else knew what they were doing either) who thought he was running "`rm -rf *`" in a subdirectory he wanted to clean out. However, he was really running this command in the *root* directory (i.e., in "/") — when he was root!** (It took a number of weeks for the department to recover since he was also in charge of backups).

1.6.16 File and directory permissions

All files and directories under UNIX have associated access permissions that can be individually set. This is necessary so that you can give other users full access, partial access, or no access to your files. There are three types of access: read, write, and execute. For files, these do what you would expect. Read access means that you can view the file, write access means that you can write to the file (or edit the file), and execute access allow you to execute the file (if it is an executable program).

Directories work similarly: read access allows you to view the contents of the directory (but not using wildcards), and write access allows you to create files in the directory. However, execute permission for a directory has a different meaning: it allows you to view the contents of the directory using wildcards or to change to that directory using the cd command. To explain all this we return to our previous long listing, but only include the specific files we want to discuss.

```
-rwxr-xr-x    1 zola  guest    13568 Jan 16 11:48 check
-rw-r--r--    1 zola  guest      123 Jan 16 11:47 check.c
drwxr-x---    2 zola  guest     4096 Jan 16 11:53 p_1
drwx------    2 zola  guest     4096 Jan 16 11:52 p_2
lrwxrwxrwx    1 zola  guest       12 Jan 16 11:54 scan.dat -> p_1/scan.dat
```

Every file or directory on a UNIX system has three sets of permissions. Recall that the first character in the listing gives the type of file. The next three sets of three characters determine the protection:

- The first set of three characters is used to determine what the owner of the file can do (denoted by "`u`" for user).

- The second set of three characters applies to users who are in the same group as the owner of the file (denoted by "g" for group).
- The third set of three characters applies to everyone else on the system, usually known as *other* (and denoted by "o").

In each set the first character denotes read access (i.e., "r"), the second character denotes write access (i.e., "w"), and the third character denotes execute access (i.e., "x"). If access is denied, replace the appropriate letter by "-".

For ordinary files the access means (for either the user, the group, or the world):

`r` – read access is allowed, i.e., you can view or copy the file.

`w` – write access is allowed, i.e., you can modify the file.

`x` – execute access is allowed, i.e., the file is an executable file and you can execute it.

For directories the access means something different and is, in fact, **very** confusing. Rather than trying to explain what each access means, we will only discuss "---", "r-x", and "rwx".

`---` – you have no access to the directory.

`r-x` – you can view any or all the files in the directory. Your access to a file in the directory is governed by the file permission of that file. You cannot create new files or delete existing files in the directory.

`rwx` – in addition to "r-x" you can also create new files or delete existing files in the directory.

Changing permissions

The `chmod` command can be used to change the permissions of files and/or directories. The mode can be represented in two ways: the absolute mode and the symbolic mode. In the absolute mode, "r" is the number 4 (i.e., 100 in base 2), "w" is the number 2 (i.e., 010_2), and "x" is the number 1 (i.e., 001_2). Thus, "-wx" is the number 3 (i.e., 011_2), "r-x" is 5 (i.e., 101_2), "rw-" is 6 (i.e., 110_2), and "rwx" is 7 (i.e., 111_2). In this way each set of permissions is reduced to a single number. For example,

```
chmod 740 filename
```

makes the file readable, writable, and executable by the owner and makes it readable by group members.

You can also represent the mode symbolically, but the syntax is much more complicated. Rather than trying to explain it, we will give a few examples. First, you denote who the permission applies to by using

`u` – owner of the file

`g` – group

`o` – other

`a` – all users

The command

```
chmod go+wx prob1.f
```

adds write and execute access to the owner of the file `prob1.f` and the group associated with the file. (Note that if either the owner and/or the group already had read access, they still do.) This does not change the permission of others (i.e., "o").

Changing the command slightly

```
chmod go=wx prob1.f
```

gives precisely write and execute access to the owner of the file `prob1.f` and the group associated with the file. (Note that if either the owner and/or the group already had read access, they have lost it.) This does not change the permission of others (i.e., "`o`").

You can also reduce access. For example,

```
chmod go-r prob1.f
```

takes away read access for the user and the group associated with the file. (If they did not have read access beforehand, nothing is changed.)

Executable access

An executable file is only executable if the execute access is set. It occasionally happens that you have obtained an executable file from somewhere else and the execute access is not set. In this case UNIX will not allow you to run the program. You will get a very strange error message such as

filename: `Permission denied.`

This cryptic error message indicates you are not allowed to do what you want. Typing

`ls -l` *filename*

will show you that the execute access is not set. You will have to add execute access to be able to execute the file.

1.6.17 grep

The *grep* utility (short for "global regular expression print") allows you to search through any number of files for all lines that match a specified string of characters or a pattern; it outputs any line that matches the string or pattern. It is commonly used when you cannot remember which file contains a certain piece of information. If you can remember an unusual word or phrase that is in the desired file, you can search for that string. For example,

`grep` *string pathname . . .*

or

`grep '`*string*`'` *pathname . . .*

searches all the files in *pathname . . .* for the specified string. The string needs to be enclosed in single quotes if it contains any spaces. The search is case sensitive unless you use the option "`-i`". In addition, you can search recursively for a string beginning in the current directory by

`grep -r` *string* .

or in another directory by

`grep -r` *string directoryname*

However, you can also search for a large number of different strings all at once by using a *regular expression*. This uses special characters to represent a large number of characters. Since there is an entire book written on "regular expressions", we will only discuss the tip of the iceberg. (One reason that it requires an entire book is that "regular expressions" evolved as UNIX evolved and different tools have slightly different "regular expressions".) In particular, "regular expressions" are not wildcards as we described them in subsection 1.5.4.

We will not try to explain "regular expressions" in detail, but only include the most common usage and some examples. (Again, the wildcards for `grep` are completely different from those discussed in 1.5.4 for `ls`, `mv`, etc.)

- `.` Match any single character. For example, `b.d` matches the character "b", followed by any character, followed by a "d", so it would match "bad", "bed", "bid", etc., but it would not match "bard".
- `*` A single item (e.g., a single character or a `[ ]`) followed by `*` matches zero or more occurrences of that character. For example, `be*n` matches "bn" or "ben" or "been", etc. `x[0-9]*` matches "x" or "x3" or "x39", etc.
- `+` The same as `*` except that it matches one or more occurrences of that character. For example, `be+n` matches "ben" or "been", etc. `x[0-9]+` matches "x3" or "x39", etc.
- `[ ]` This works the same as we discussed in subsection 1.5.4.
- `\` Converts a special character to a normal character. For example, to search for a "*", use "*".

1.6.18 find

The *find* utility is a very complicated command which finds files or file names that match any of a large number of conditions. (For example, files which are at most a week old or files that belong to a specific user or file names that match certain wildcard constructions.) The reason it is such an important utility is that it finds files not only in a particular directory (usually `.`), but also **in all subdirectories** of that directory. It is much too complicated to discuss here, except for one simple use. If you are looking for a file with a specific name — or maybe you only remember part of the name — the `find` command is what you want, if only you can remember how to use this command. Most likely, you will only use this command very infrequently, and you will forget *how* to use it. So here we show a very simple use, namely

```
find .
```

This prints out *all* the file names in the current directory and any and all subdirectories. If you pipe this into the `grep` command, i.e.,

```
find . | grep string
```

you will obtain all the file names that **include** the characters *string*. For example,

```
find . | grep tex
```

will find any and all files which include the characters "tex". This will include file names such as "unix.tex", "textbook.arg", and "context_is_everything.txt".

Well, ok, if you really want to know how to use the `find` command directly to find files, here it is. Enter

```
find . -name 'string'
```

where *string* can include any of the wildcards discussed in 1.5.4. For example,

```
find . -name '*.tex'
```

will find all files which end in the extension "tex" and

```
find . -name 'a?.f'
```

will find all files which have two characters preceding the "." with the first being an "a" and the where extension is "f".

1.6.19 Common file extensions

A filename usually contains three parts: a name, a dot, and an extension. The extension identifies the type of file. Many of these file types indicate text files, which are often called ASCII files. A quick list of popular file extensions and what program(s) can be used to view them (if they are not ASCII files):

`.ps` Postscript file. View with `gv`.

`.eps` Encapsulated postscript file. View with `gv`.

`.pdf` Adobe Acrobat file. View with `acroread`.

`.jpg` JPEG image file. View with `display`.

`.gif` GIF image file. View with `display`.

`.tif` TIFF image file. View with `display`.

`.mpg` MPEG movie file. View with `mplayer`.

`.f` Fortran source code.

`.f90` Fortran 90 source code.

`.c` C source code.

`.cpp` C++ source code.

`.o` A compiled, i.e., an object, file (see section 2.1) from Fortran, C, or C++. Cannot be viewed.

`.a` A library file (see section 2.1). Cannot be viewed.

`.java` Java source code.

`.class` A byte compiled file from Java. Cannot be viewed.

`.html` An HTML file.

`.tex` A TEX or LATEX file.

`.dvi` A compiled TEX or LATEX file. View with `xdvi`.

`.zip` File compressed with `zip`; uncompress with `unzip`. Cannot be viewed.

`.gz` File compressed with `gzip`; uncompress with `gunzip`. Cannot be viewed.

`.tar` A tarred file; un-tar using `tar` as described in section 1.10. Cannot be viewed.

`.tgz` A tarred and compressed file; uncompress with `gunzip`, or untar and uncompress using `tar` as described in section 1.10. Cannot be viewed.

1.6.20 Useful subdirectories to know about

The filesystem tree shown in Figure 1.1 includes a number of useful subdirectories. Notice that there are *four* `bin` directories (short for "binary") shown — there are actually many more, but we are making a point. These directories contain executable files (which we discussed in subsection 1.6.16). `/bin` contains most of the "basic" UNIX tools whereas `/usr/bin` contains many of the UNIX applications. `/usr/local` is where the system administrator usually adds applications which do not come with the UNIX distribution so `/usr/local/bin` is where the "local" executable files are contained. In addition, the directory `smith` also contains the subdirectory `bin`. This subdirectory is where, by convention, you would put any "shell-like" programs that you might write. How does UNIX know where to look for executable programs? By using the `PATH` environment variable, which we discuss in subsection 1.9.2.

Note that there are four `lib` directories (short for "library") shown. If you are looking for a particular library (see section 2.1), `/lib`, `/usr/lib`, and `/usr/local/lib` are good places to start. If you want to link a particular library to your program, you have to do it yourself.

1.7 Redirecting Input and Output

When a program runs on a UNIX system, it usually expects some input and produces some output. By default, a program will use the *standard input* (often called `STDIN`), which is usually the keyboard, and the *standard output* (often known as `STDOUT`), which is usually the screen. In addition, if the program reports any errors encountered during execution, it uses *standard error* (often called `STDERR`), which is also usually the screen.

A program can be told where to look for input and where to send output, using input/output redirection. UNIX uses the "<" (the "less than" character) to signify input redirection, and and ">" (the "greater than" character) to signify output redirection.

Probably the most common use of redirection is to save the output of some commands to a file, rather than have it appear on the screen. For example, the command

`ls` > *filename*

redirects the output of the `ls` command to the file *filename*. If the file to the right of ">" already exists, then the command aborts and tells you that the file already exists. There are several variations of redirection that you can use to redirect standard output:

> *filename*	redirect output to *filename*, abort if *filename* exists.
>! *filename*	redirect output to *filename*, overwrite *filename* if it already exists.
>> *filename*	append output to *filename*, abort if *filename* does not exist.
>>! *filename*	append output to *filename*, create *filename* if it does not exist.

For example, to append the contents of *filename2* to *filename1* you can use the append redirection ">>" and the `cat` command, such as

`cat` *filename2* >> *filename1*

The cat command normally shows the contents of *filename2* on the screen. However, the redirection command causes the contents to be appended to the file *filename1*. The command is aborted if *filename1* does not already exist.

Redirecting standard input works similarly, using the "<" operator (you can keep them straight by remembering which way you want the data to flow). Input redirection is not widely used. However as a simple example, if you have written a program which requires input from the keyboard, you can put the data in a file and use "<" to change standard input from the keyboard to this file.

Another type of redirection is called *piping*. Piping is when the output of one command is redirected to become the input of another command. The pipe command is a "|" (the vertical bar). A common example is:

```
ls -al | less
```

This takes the output of the command "`ls -al`" (the command to list all files in a directory in long format) and sends the output to the input of `less` (which displays data one page at a time). When you do this, the file listing is passed to the tool `less`, which allows you to paginate through the files rather than having to scroll back to see the files at the beginning.

1.8 Printing

You should use the command `print` to print your files.

Note: The standard print command in UNIX is `lpr`. The print command `print` was written locally and does not exist in standard UNIX. See subsection 1.8.1 for more details.

1.8.1 Printing a File

Printing files is done with the command

`lpr` *filename . . .*

This will print the files you specify to the printer defined by the environment variable `PRINTER` (see 1.9.2). The default is that printing is done on both sides of the paper. `lpr` accepts a number of options, some of which are the following:

`-o`	various options, such as `sides=one-sided` or `Duplex=None` might print single-sided
`-P`	specify which printer to use

For example, the command

```
lpr -P mw502 thesis.tex
```

will print to the printer in MW 502.

`lpr` is currently capable of printing the following types of files:

- plain text files
- pdf files (i.e., .pdf)
- Postscript (i.e., .ps or .eps)
- gif files (i.e., .gif)
- tiff files (i.e., .tif)

If your document has one of these extensions, it will be printed correctly.

1.8.2 Querying the Printer

When you print a document, it is added to the *print queue*. Items in the queue are printed in the order they were added, so if other items are ahead of your job they will be printed first.

You can check the status of a printer with the command

```
lpq
```

This will list any current jobs waiting to print, or the last printed job if it is currently idle. You can use the "`-P`" flag to check the status of other printers.

1.8.3 Removing Print Jobs

If you print something and then decide you would like to cancel it, you can try to use the command

`lprm` *jobnumber*

If the job is not on your default printer, you will need to specify the printer using the "`-P`" flag. However, this rarely works since once the document has been sent to the printer, it is removed from the queue and is stored in the printer's memory.

1.8.4 Printer Etiquette

While we do not impose any limitations on the amount of documents you can print, you should be considerate of other users:

- If you have a very large document to print, try to do it during off-peak hours (such as in the evening).
- Cancel any print jobs that you enter by mistake.
- Also, if you need multiple copies of a document, please only print out **one** copy. Use the xerox machine or the duplication facility to make additional copies.

- Get your printout as soon as possible. This will save others from having to thumb through your output, and it will save you from having others take your output by mistake.
- Finally, we ask that all users keep the printer room clean by placing unwanted printouts in the recycling bin.

1.9 Advanced Shell Commands

When you log in to a UNIX system, the kernel starts a shell for you, and connects the shell to your terminal. When you execute a command from the shell, the shell creates a child process to execute the command and connects the child process to your terminal. By connecting the child process to your terminal, the shell allows you to send input to the child process and receive output from it. When the child process finishes, the shell regains access to the terminal, redisplays the shell prompt, and waits for your next command.

Any task that requires you to actively participate (like word processing) must be in the foreground to run. Such *interactive* jobs must periodically update the display, and accept input from you; this requires access to the terminal interface. *Noninteractive* jobs do not require you to participate once they are started. For example, a job that sorts the contents of one file and places the results in another file would not have to accept user commands or write information to the screen while it runs. Some UNIX shells allow such noninteractive jobs to be disconnected from the terminal, which frees the terminal for further interactive use. (It is even possible to log out and leave a background process running; Unfortunately, there is no way to reconnect a background job to the terminal after you've logged out.)

Jobs that are disconnected from the terminal for input and output are called *background* jobs. You can have a large number of background jobs running at the same time, but you can only have one *foreground*, i.e., interactive, job. That is because you only have one screen and one keyboard at your terminal.

1.9.1 Background and foreground jobs

A UNIX process can be disconnected from the terminal and allowed to run in the background; this is called a *background* job. Because background jobs are not connected to a terminal, they cannot communicate with the user. If the background job requires interaction with the user, it will stop and wait to be reconnected to the terminal.

When you run a command, by default it is started in the foreground. You can start it in the background by following the command with an ampersand, i.e., an "`&`". For example, the command

```
mozilla &
```

launches the mozilla web browser and switches it to the background. You can now continue using the shell to do other work. Since mozilla is a graphical X-Windows application (so it does not require any input from the keyboard), you can use mozilla's graphical interface even though its job is backgrounded. Without the "`&`", mozilla would continue to run in the foreground and you would be unable to use the shell to do anything else until you killed mozilla.

You can view all running jobs by typing

```
jobs
```

which lists all the current jobs and their status. As an example, the output of this command might be

```
[2]  - Running     mozilla
[3]  + Running     emacs
```

There are two jobs running in the background: the mozilla web browser and the emacs text editor. Mozilla is job #2 and emacs is job #3. (In this case there is no job #1.) The last backgrounded job is #3, which is indicated by the "`+`".

Suppose you want to bring one of these jobs to the foreground. Perhaps you were editing a document in emacs and you suspended that job (we'll cover that shortly) and now you want to bring it back. You do this with the command

```
fg
```

(short for "foreground"). This brings the last backgrounded job to the foreground. If you would like to foreground a job other than the last one, simply add the job number. For example, the command

```
fg 2
```

will bring mozilla to the foreground.

Now that we've discussed how to bring a job to the foreground, how can we send one to the background? This is very easy as well. Type `C-z` to suspend the process; that is, the process is now *frozen* and will not respond to any commands. When you do this, the shell outputs the job number, the process, and the status as "Stopped" or "Suspended". Control is now returned to the shell. To bring the program back to life, you need to send it to the foreground (as described above), or to the background. The command

```
bg
```

backgrounds the just suspended job. You can now run other programs.

Warning: **Do not use `C-z` to kill jobs — it doesn't kill them, it merely suspends them. This is a very, very bad thing to do!!! Use `C-c` instead.**

You can also kill a process by typing

`kill` *processnumber*

where *processnumber* is the number of the process, which is found by using the `jobs` command described above.

On a final note, programs that output data will still output to standard out (i.e., `STDOUT`) even while they are backgrounded. This means that while you are interacting with another program the backgrounded program can still write to the screen. To prevent this, you would launch the job with the syntax

```
emacs > /dev/null
```

(We have already discussed the redirection command ">" in section 1.7.) This sends the output to the file `/dev/null`, which is the equivalent of a black hole on a UNIX system.

Note: In UNIX there are two separate outputs: STDOUT (standard out) and STDERR (standard error). Only the "normal" output of the program is being sent to a black hole. Any error messages will still appear on the screen — and likely "mess up" the screen. (You can use `C-l` to clear the screen). You can output *both* STDOUT and STDERR to the same file by ">&" so that, for example,

```
emacs >& /dev/null
```

will send both outputs to the black hole.

1.9.2 Bash vs. tcsh

The two shells we support are *bash* (short for "the Bourne Again Shell") and *tcsh* (an enhanced C-Shell). Most of the key commands are similar — but not necessarily identical. The differences will be pointed out during the following discussion on useful shell concepts.

Aliases

One of the more useful features of a shell is that you can define aliases for commands. For example, you can define an alias for `rm` so that it actually executes "`rm -i`". In this way, you know exactly what files you are deleting. In fact this is the default in both bash and tcsh. Also, `mv` has been aliased to "`mv -i`" and `cp` has been aliased to "`cp -i`".

Both bash and tcsh support aliases, but with a slightly different syntax. The following assigns the command "`rm -i`" to the alias `rm`.

bash: `alias rm=rm -i`
(*Note that there are no spaces around the "=".*)

tcsh: `alias rm 'rm -i'`

You can get a list of currently set aliases by typing

```
alias
```

If you would like to save your aliases so that they are automatically set every time you log in, add these commands to `.bashrc` for the bash shell or `.cshrc` for tcsh.

Completions

Both shells also offer command and path name completion. This allows you to type a partial command or path name and have the shell automatically attempt to complete it. If the partial name is unique it is completed; otherwise, the shell waits for you to enter more characters. (This is *fantastic* for long file names.) Again, this works slightly differently for each shell.

In bash, completions are handled by using <`TAB`>. If you type `ema` and hit <`TAB`>, the shell will finish typing `emacs` for you (assuming no other commands begin with "ema"). If there are multiple matches, hitting <`TAB`> one or more times again in succession will show you all possible matches. Simply type a few more letters of the command or the path name and enter <`TAB`> again; you will be closer to total completion.

In tcsh, completions are also handled by using <`TAB`>. However, if there are multiple matches, type `C-d` to see all possible matches.

Environment Variables

Each shell has several important environment variables necessary for everyday use. Environment variables define certain things that enhance and customize each shell to your own taste. Some of the more important variables follow.

`PATH` This variable contains a colon separated list of directories to use in searching for a command.

`HOME` This variable contains your home directory. There are a number of uses for this command, but the only one we will discuss is in the section on the tool make, (see section 4.1).

`EDITOR` This variable is very useful. If you prefer a certain text editor over another, then this variable should be set. The editor of choice must be installed. For example, mutt uses the `EDITOR` variable to decide which text editor is used when composing email messages. By default, `EDITOR` is set to vi, but if you prefer to use emacs, then set `EDITOR` to emacs.

`PRINTER` This defines your default printer so that you don't have to specify a printer every time you send a file to the printer.

In both bash and tcsh you set environment variables in the file `.login`. To set an environment variable, type

bash: `export` *ENVIRONMENT_VARIABLE=NewValue*
(*Note that there are no spaces around the "=".*)

tcsh: `setenv` *ENVIRONMENT_VARIABLE NewValue*

You can always determine the value of an environment variable by entering

`echo` *$ENVIRONMENT_VARIABLE*

The "`$`" is necessary before the name of the environment variable or else the echo command will simply echo back the name of the argument. You can also see all the environment variables which are set by typing

bash: `set`

tcsh: `printenv`

Odds and ends

You can clear the screen by typing `C-l`. This corresponds to the UNIX command `clear`.

Both shells keep track of the commands you have entered. Type `history` to see a list of your most recent commands. You can recall a previous command in a number of ways:

- Type ↑ to recall each previous command in turn. After recalling a previous command, you can then type ↓ to recall each following command in turn. You then type <`ENTER`> to execute the command.
- Type `C-p` to recall each previous command in turn. After recalling a previous command, you can then type `C-n` to recall each following command in turn. You than type <`ENTER`> to execute the command.
- Type `!!` to execute the previous command.
- Type `!`*string* to execute the most recent command which *begins* with *string*.
  Note that *string* need not be the entire command. For example, `!s` will execute the last command which began with the letter "s".
- After you have typed `history` to see a list of your most recent commands and found the number corresponding to the command you want, type `!`*number* to execute that command.

You can also edit the command you are presently working on using emacs commands:

← backspace one character.

`C-b` backspace one character.

→ go forward one character.

`C-f` go forward one character.

`C-a` go back to the beginning of the line.

`C-e` go forward to the end of the line.

Other commands that work include `C-d`, `C-SPACE`, `C-x C-x`, `C-k`, `C-w`, and `C-y`.

Below is a table summarizing the commands we have discussed.

Useful shell commands		
description	bash	tcsh
alias command	`alias` *new_name*=*command* (*No spaces around "="*)	`alias` *new_name command*
Print all aliases	`alias`	`alias`
Set environment variable	`export` *ENV_VAR*=*value* (*No spaces around "="*)	`setenv` *ENV_VAR value*
Print all environment variables	`set`	`printenv`
Where to store your aliases	`.bashrc`	`.cshrc`
Where to store your environment variables	`.login`	`.login`
Command and pathname completion	<TAB>	<TAB>
Print multiple matches	<TAB><TAB>	`C-d`
history	print out the last commands you entered	
Execute last command	`!!`	
Execute last command that begins with *string*	`!`*string*	
Execute command number *number* (use `history` to find numbers of commands)	`!`*number*	
Recall each previous command in turn	↑ or `C-p`	
Recall each following command in turn	↓ or `C-n`	

1.10 Compressing, Tarring, and Transferring Files Between Computers

1.10.1 Compressing files

Files can take up lots of space in a file system. If they are text files, you can compress them so that they take up less room. A standard tool to do this is `gzip` (short for "GNU zipper"). For example, you can compress any number of files by

`gzip` *filename ...*

You can compress all the files in the current directory and in all subdirectories by

```
gzip -R *
```

(Directories themselves are not compressed.) The extension `.gz` is added to all file names. To uncompress `.gz` files, use `gunzip`.

1.10.2 Tarring files

You can combine a number of files, even if they are in a number of different directories, into one file. This is normally done to copy the files from one computer to another. Many software packages are also tarred for easy downloading. The tool which does this is `tar` (short for "tape archiver"). This tool has been in UNIX for a long time and it is *very* user unfriendly.

To create a tar file, enter

`tar cf` *tar_pathname*`.tar` *pathname ...*

(Normally, you put the extension `.tar` on *tar_pathname*.) "*pathname ...*" can be one or more files (including wildcards) and/or one or more directories (including wildcards). To untar a tar file, type

```
tar xf tar_pathname.tar
```

To remember the options: first, you have to create (`c`) the tar file (`f`), then you have to extract (`x`) the tar file (`f`).

After creating a tar file, it is often worthwhile to compress it. Some versions of `tar` also allow you to gzip and gunzip files "on the fly" by

```
tar czf tar_pathname.tgz pathname ...
```

(Normally, you should put the extension `.tgz` or `.tar.gz` on *tar_pathname*.) To untar a compressed tar file, type

```
tar xzf tar_pathname.tgz
```

1.10.3 Transferring files between computers

To transfer files between computers you use `sftp` (short for "secure file transfer protocol"). The common terminology is that you are running on one computer, the *local computer*, and you want to transfer files to and/or from another computer, the *remote computer*. If your username is the same on both computers, type

```
sftp remote_computer
```

while if your username is different, type

```
sftp remote_username@remote_computer
```

where *remote_username* is your username on *remote_computer*. You will be prompted for your password on *remote_computer* and then you will be connected.

The simplest steps needed to transfer files are the following:

1. cd to the directory you want to be in on the local computer.
2. Connect to the remote computer using `sftp`.
3. cd (in `sftp`) to the directory you want to be in on the remote computer.
4. Use `put` to transfer one file at a time from the local computer to the remote computer and/or `get` to transfer one file at a time from the remote computer to the local computer.

The "essential" commands are discussed in more detail below. (**You are not allowed to use wildcards in any of these commands.**)

`help` Shows you all the commands and a *very* brief explanation of each.

`cd` "`cd` *directoryname*" changes directories on the remote computer.

`put` "`put` *local_filename*" puts a *single* file on the remote computer using the same name. "`put` *local_file remote_file*" puts a *single* file on the remote computer using a different name. **Warning: If there is a file of the same name on the remote machine, it will be overwritten without any warning.**

`get` "`get` *remote_filename*" gets a *single* file from the remote computer using the same name. "`get` *remote_file local_file*" gets a *single* file from the remote computer using a different name. **Warning: If there is a file of the same name on the local machine, it will be overwritten without any warning.**

`exit` Exit or quit `sftp`.

`quit` Exit or quit `sftp`.

The utility sftp actually contains about 20 commands. Once you are connected you can change directories locally or remotely and list all the files in the current directory. (**You are not allowed to use wildcards in any of these commands.**)

`lcd` "`lcd` *directoryname*" changes directories on the local computer.

`pwd` Prints the current working directory on the remote computer.

`lpwd` Prints the current working directory on the local computer.

`ls` Lists all the files in the current working directory on the remote computer.

`lls` Lists all the files in the current working directory on the local computer.

In addition, you can make and delete directories, delete files, and rename files on the remote computer. (**You are not allowed to use wildcards in any of these commands.**)

`mkdir` "`mkdir` *directoryname*" creates a new directory on the remote computer.

`rmdir` "`rmdir` *directoryname*" removes a new directory on the remote computer.

`rename` "`rename` *remote_oldfilename remote_newfilename*" renames a *single* file on the remote computer.

`rm` "`rm` *filename*" removes a *single* file on the remote computer.

One annoying feature of sftp is that you can only transfer *one* file at a time. Thus, it is common for a user to tar all the files to be transferred and then only have to transfer the one tarred file. Possibly, there is an alternative. Some versions of sftp have two extra commands which allow you to transfer multiple files (i.e., to use wildcards). These two commands are `mput` (i.e., "multiple `put`") and `mget` (i.e., "multiple `get`). Unfortunately, even if your version of sftp has these commands, they might not be listed when you do `help` or when you do "`man sftp`". You just have to try them and see if they work. (**You <u>can</u> use wildcards in any of these commands.**)

`mput` "`mput` *local_filename* ..." puts any number of files on the remote computer.

`mget` "`mget` *remote_filename* ..." gets any number of files from the remote computer.

1.11 Transferring Files Between Operating Systems

Someday you may well need to transfer files between UNIX, Microsoft Windows, and/or Macintosh computers. For example, you might be using a UNIX computer or a Macintosh computer in the department and have a Microsoft Windows computer at home or on your laptop. You might have written a text document at home or on the road and want to upload it to the department to continue working on it. A difficulty that often arises when you transfer text files is that each operating system has a different convention for ending a line:

- In UNIX the end of line character is a carriage return (i.e., `C-J`).
- On a Maxintosh the end of line character is a line feed (i.e., `C-M`).
- And in Microsoft Windows the end of line character is actually two characters, a carriange return followed by a line feed (i.e., `C-J C-M`).

Frequently, a file creating using one of these formats will appear corrupted when accessed by another and so the format must be changed. This can be easily done in emacs on any operating system. See subsection 3.1.8 for details.

In UNIX you can change between the Microsoft Windows and the UNIX formats, but not to or from the Macintosh format. (However, you can change between all of these using emacs; see 3.1.8.) To

change the end of line character from Microsoft Windows to UNIX, use the UNIX command `dos2unix`, and to change the end of line character from UNIX to Microsoft Windows use the UNIX command `unix2dos`,

In the "good old days" everything would now work fine because a "text file" meant a file using the ASCII printable character set. This was back in the days when everyone used English — even if they didn't know any English. However, nowadays there are many character codes in use representing many, if not most, of the languages in the world. We discuss some of these character codes below.

Warning: You can only reliably transfer text files between operating systems. However, you cannot transfer executable files between operating systems. You may not be able to transfer internal files from application such as word processors between operating systems.

1.11.1 The ASCII character set

When you type characters into the computer they are stored in the computer in *bytes*. (One byte equals eight bits, where each *bit* can store the number 0 or 1.) Each character has a different representation and so there are 256 different charactes that can be represented in the computer.

The *character code* defined by the *ASCII* (short for, American Standard Code for Information Interchange) standard has been around for a number of decades. Although there are 256 possible characters codes (i.e., from 0 to 255), not all of them are used. The printable characters are the following:

```
  ! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
```

where the character in the top-left corner is the blank character. Code values are assigned to these characters in the order in which the characters are listed above from left to right and then from top to bottom. The first character (i.e., the blank character) is assigned the number 32 and the last character (i.e., the tilde) is assigned the number 126. Positions 0 through 31 represent control characters with `C-A` through `C-Z` assigned to the number 1 through 26. The backspace character is `C-H`, i.e., number 8, and the delete character is number 127. The numbers 128 through 255 are not used.

ASCII is often denoted by US-ASCII nowadays. There are many other character sets, such as *ISO 8859-1* through *ISO 8859-16*, which represent character codes for languages throughout the world. If you try reading a text file using one character code but which generated using another, you will get gobbledygook.

In order to encompass *every* character code into one, *ISO 10646*, common called *Unicode*, was created. It represents one caracter using <u>2</u> bytes, and so can represent 65,536 characters.

1.12 Summary of Commands

Below we provide a summary of the commands, tools, and applications we have discussed in this chapter; the phrase inside "{ }" might help you remember the name of the command. The only command not yet discussed is `script` (under "Miscellaneous). It is useful for students to keep a record of their work.

Files and Directories	
`cd`	{"change directories"} Change your current directory. (See 1.6.3.)
`chmod`	{"change mode"} Change file directory access permissions. (See 1.6.16.)
`cp`	{"copy"} Copy files and/or directories. (See 1.6.10.)
`find`	Find files or file names that match certain conditions in a particular directory and in any of its subdirectories. (See 1.6.18.)
`ln`	{"link"} Create a symbolic link to a file. (See 1.6.6.)
`ls`	{"list"} List files and/or directories. (See 1.6.4.)
`mkdir`	{"make directory"} Make new directories. (See 1.6.13.)
`mv`	{"move"} Move and/or rename files or directories. (See 1.6.9.)
`pwd`	{"print working directory"} Print the current directory. (See 1.6.3.)
`rm`	{"remove"} Remove files. (See 1.6.11.)
`rmdir`	{"remove directory"} Remove directories. (See 1.6.15.)

Logging in to another computer and transferring files between computers	
`sftp`	{"secure file transfer program"} Transfer files between computers. (See 1.10.3.)
`ssh`	{"secure shell"} Access another computer. (See 1.2.)

Miscellaneous	
`apropos`	{"manual"} Use keywords to find UNIX commands related to the keywords. (See 1.5.5.)
`ar`	{"archive"} Create and maintain a library of object files. (See 2.1.)
`clear`	Clear the screen. The shell command `C-l` does the same thing. (See 1.9.2
`echo`	Display on the screen whatever follows this command. (See 1.9.2.)
`exit`	Logout. (See 1.3.4.)
`man`	{"manual"} View manual pages. (See 1.5.5.)
`mozilla`	a web browser (See 1.4.)
`phone`	a simple way to access our departmental phone list (See 1.4.) **This is a local command — not a standard UNIX command.**
`passwd`	{"password"} Change your password. (See 1.3.3.)
`script`	Copy everything that is output to the screen into a file; the default file name is `transcript`.

Viewing files	
`cat`	{"concatenate"} Concatenate files or view short files. (See 1.6.8.)
`grep`	Search files for all lines that match a particular pattern. (See 1.6.17.)
`head`	View the head (i.e., the first part) of a text file. (See 1.6.8.)
`more`, `less`	View a text file. (See 1.6.8.)
`tail`	View the tail (i.e., the last part) of a text file. (See 1.6.8.)

Shell Commands	
`bg`	{"background"} Move a process to the background. (See 1.9.1.)
`fg`	{"foreground"} Move a process to the foreground. (See 1.9.1.)
`jobs`	View all processes. (See 1.9.1.)
`kill`	Kill a process. (See 1.9.1.)

Print	
`lpr`	{"line printer"} The standard UNIX print command. (See 1.8.1.)
`lpq`	{"line printer queue"} Show the printer queue status. (See 1.8.2.)
`lprm`	{"line printer remove"} Remove a print job by "`lprm` *jobnumber*". (See 1.8.3.)

Compressing and transferring files	
`gunzip`	{"GNU unzip"} Uncompress files. (See 1.10.1.)
`gzip`	{"GNU zip"} Compress files. (See 1.10.1.)
`tar`	{"tape archive"} The primary use is to combine files into a single file (called an *archive*) for easy storage and distribution. (See 1.10.2.)

Transferring files between operating systems	
`dos2unix`	Convert file from Microsoft to UNIX (See 1.11.)
`unix2dos`	Convert file from UNIX to Microsoft (See 1.11.)
Note: in emacs you can switch between unix, dos, and mac (See 3.1.8).	

Chapter 2

Languages

In this chapter we discuss how to compile (or interpret) and run your programs. This discussion is limited to Fortran, C, C++, and TEX and LATEX.

2.1 Compiling and Running Your Fortran, C, and C++ Programs

When you write a program in C, C++, Fortran 77, or Fortran 90, you write the source code using an editor such as emacs (see section 3.1) or vi (see section 3.2). In order to execute the program, you have to create an executable (or a binary) file. You do this by running the compiler. The compiler actually carries out two steps:

1. it compiles the source code and generates an object file;
2. and it loads the object file and a number of libraries to create the executable file.

(You can combine these two steps into one so that object files do not appear in your current working directory; however, they have still been created.) The result is also called "machine code" and consists of 0's and 1's that the computer's CPU understands.

Why are there two steps? Normally, programs are broken up into "pieces", which we will call *subprograms*. (You might know these under the names "functions" or "subroutines" or "procedures" or "modules".) If you have a number of subprograms in your source code, you should split them up into a number of different files. One very important reason for this is that you can compile each file separately. If a file compiles, fine; if not, you do not have to search through hundreds or thousands of lines to find your errors. And later, after you have debugged a file and are convinced it is correct, you compile the file and never have to touch it again. (In fact, in Section 4.1 we will discuss a UNIX tool which will compile and load any number of separate files, and it will only recompile a file if you have changed it.)

The same is true of the compiler. In your program you often input and output data, convert data from one type to another (such as from integer to double precision), use mathematical functions such as the cosine or the arctangent, and call subprograms which are part of the language. All of these require subprograms which have been written by whomever wrote the compiler. It would be very inconvenient to have to work with each of these object files. Instead, this large number (hundreds or thousands) of object files are combined into a small number of library files. In UNIX this is easily done by

`ar c` *library_filename filename . . .*

Normally, you do not have to worry about including these libraries because the compiler automatically includes them. However, see the warning below if you use C or C++.

In addition, you sometimes have to include other packages in your program. For example, to solve the linear system $Ax = b$ where A is an $n \times n$ nonsingular matrix, you can use LAPACK. This package contains hundreds of separate files. It would be very difficult if you had to include the necessary files

yourself. Instead, all the files have been collected into the file `liblapack.a` and put in the directory `/usr/local/lib` .

When you have compiled all your source files, you are ready to load the program. The loader loads your object files as well as any and all libraries to form your executable code. Some libraries are automatically included by the loader; others you have to explicitly include yourself.

Now, let's get down to details. Here are the various compilers we have in Linux.

languages	**Linux**
C (.c)	cc and gcc are the same (GNU)
C++ (.cpp)	c++ and g++ are the same (GNU)
	icc (Intel compiler)
Fortran 77 (.f)	f77 and g77 are the same (GNU)
	(and are actually gfortran)
Fortran 95 (.f90)	gfortran (GNU)
	ifort (Intel compiler)

To *compile* a file, type

compiler `-c` *filename.ext*

where *compiler* is the appropriate compiler command and *ext* is the appropriate extension. The output is a file with the extension `.o`.

compiler `-c` *filename.ext*

To *compile and load* a file, type

compiler *filename.ext*

The output is the executable file `a.out`. That is, the default name of *every* executable file is `a.out`. This is not very clever (but this comes from the *very* early days of UNIX). The most common convention is to use the same filename for your executable file, but without any extension. You do this by

compiler `-o` *filename* *filename.ext*

That is, the option "`-o`" requires a file name following it, and that file name is the name of the executable file.

Suppose we have three files, `file1.c`, `file2.c`, and `file3.c`, where the main routine is in `file1.c`. We can compile and load these files separately by

```
cc -c file1.c
cc -c file2.c
cc -c file3.c
cc -o file1 file1.o file2.o file3.o
```

or we can compile and load all of these files at once by

```
cc -o file1 file1.c file2.c file3.c
```

or we can mix and match by, for example,

```
cc -c file2.c
cc -o file1 file1.c file2.o file3.c
```

We can now execute this program by simply typing

```
file1
```

(This command will normally work, but read subsection 2.1.3.)

Next, let us discuss libraries and suppose that `file1.c` contains calls to LAPACK subprograms. To include the LAPACK library `liblapack.a` you merely add "`-llapack`" at the end of your load command by

```
cc -o file1 file1.o file2.o file3.o -llapack
```

That is, all files are named `lib`*libraryname*`.a` and you include the library by adding `-l`*libraryname* at the end of the compile line.

Warning: *There is no space between the* "`-l`" *and the library name.*

Warning: One problem with the C and C++ compilers from GNU is that it does not automatically include the math library. For example, the following code, let's call it `try.c`,

```
#include <stdio.h>
#include <math.h>
int main() {
  double x = sin(1.);
  printf("%e\n", x);
}
```

will compile but will not load. It will return with a strange error message stating that it cannot find the function "`sin`". The solution is to explicitly include the math library `libm.a` by adding "`-lm`" **at the end of the load command**. In our example, we would compile and load the program by

```
cc -o try try.c -lm
```

2.1.1 Flags

There are a number of options you can use to modify the compiling and/or loading of your program. You have already seen the "`-c`" flag which compiles, but does not load, the file. Also, you have seen the "`-o`" flag, which renames the executable file.

There are many more options for each compiler — many, many more than you will ever be interested in. Two which are of general use follow.

- `-g` This adds debugging information so you can use a debugger to, appropriately enough, debug your program. This makes your program a little larger and it might run a tiny bit slower. Debuggers are discussed in detail in Chapter 4.2.
- `-O` (the capital letter O, not the number 0) This optimizes your code. The compiler tries to do everything it can to make your code run faster. Often your code will only run a tiny bit faster, but sometimes it will run a lot faster. Unfortunately, on rare occasions it many run incorrectly. Use this flag at your own risk.

2.1.2 Libraries

There is one final, **crucial** detail about libraries in the loader statement. One important function of the loader is to resolve function (or subroutine or subprogram) calls. Consider the linking statement

```
cc -o file1 file1.o file2.o file3.o -llibrary1 -llibrary2 -lm
```

The loader goes through these files, one at a time, from left to right. It begins with `file1.o` and it finds all the function calls which cannot be resolved. That is,

FINISH OFF???

2.1.3 UNIX "quirks"

In other operating systems you need to enter a command, such as `run` or `execute`, to execute one of your programs. These operating systems make a distinction between the programs (i.e., the commands) inherent in the operating system and the programs you write. For example, to list the files in a directory you might type `dir`, but to run the program `file1` you would type "`run file1`".

On the other hand, in UNIX there is no distinction between the tools in the operating system and your executable programs. They are all executed by simply typing their names.

Of course, this leads to the next question: how does the operating system know where to look to find a particular program? For example, if you type `file1`, how does UNIX know where this executable file is? The answer is that it uses the `PATH` environment variable, which we discuss in more detail in section 1.9.2. This determines exactly which directories are searched for the executable file. For example, suppose that `PATH` is set to

```
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/home/zola/bin:.
```

The colon (i.e. "`:`") separates the various directories so the shell will search these directories, in order, until the file is found. (That is, the directories will be searched in the order `/bin` , `/usr/bin` , `/usr/local/bin` , `/usr/X11R6/bin` , `/home/zola/bin` , and, finally, " `.` ", i.e., your current working directory). If no file is found, an error message will appear on the screen; if there is more than one file with the same name, the first one found will be executed. The directories listed in `PATH` are called your *path*.

Warning: The last directory in this example is "`.`", i.e., your current working directory. If this directory is not included, files in your current working directory will not be executed, because the shell will not look there. For example, if "`.`" is not included in `PATH` and `file1` is in your current working directory, then

```
file1
```

will not work. You will receive an error message similar to

```
file1:  command not found
```

However,

```
./file1
```

will work, because in the latter case you are giving the full path name of the file.
Normally, your account has been set up so that "`.`" is included in your path and so you should never have this occur on *our* computers. However, it will probably happen to you at least once in your lifetime somewhere.

Warning: There is one "feature" in UNIX that catches most users at some time in their lives.
Never name an executable file `test`.
It seems so reasonable to test some idea in Fortran or C or C++ and name the source code `test.`*`ext`* and so name the executable file `test`. You then try to execute `test` and discover something *very*, *very*, *very* strange: the program will not run! No matter what you do, the program will not run. You can see that the file exists by using ls, but the shell refuses to find it. The reason your program will not run is that there already is a program called `test` in UNIX! And, normally, it is found in your path before your program. The solution is to type `./test` and then your program will run. However, the real solution is never to name a program of yours `test`.

2.2 Running TeX and LaTeX Programs

TeX is a phototypesetting computer language. It was designed by Donald Knuth in the 1970's to be the assembly language for typesetting text — and, particularly, mathematical expressions. Since it was designed to be an assembly language, it is incredibly hard to use — but it can do almost anything having to do with text and mathematical layout.

Warning: It is important to be able to pronounce TeX correctly, so as not to be thought an uncultured

boor. TEX($\tau\epsilon\chi$) is pronounced similar to *blecchhh*[1], not to the large southern state known for "Tex-Mex" chili. (By the way, whatever *Skyline Chili* makes — it **isn't** chili.)

LATEX can be thought of as a high-level language (such as Fortran) as compared to the low-level language TEX. It is mainly used for medium-to-large technical and scientific documents and, in particular, many mathematics books. It is based on the idea that it is better to leave document design to document designers, and to let authors get on with authoring documents.

In both TEX and LATEX the filename extension is ".tex". To interpret a TEX document, type

`tex` *file*`.tex`

while to interpret a LATEX document, type

`latex` *file*`.tex`

The output is the file *file*`.dvi` (short for "device independent"). This "dvi" file can be viewed using `xdvi` or printed out using `print`. In addition, it can be converted to a Postscript file using `dvips`.

In addition, you can interpret a LATEX document by

`pdflatex` *file*`.tex`

which immediately produces a PDF file. By the way, this is the default in TeXShop on Macs.

[1]To quote Knuth, "It's the 'ch' sound in Scottish words like *loch* or german words like *ach*; it's a spanish 'j' and a Russian 'kh'. When you say it correctly to your computer, the terminal may become slightly moist."

Chapter 3

Text Editors

3.1 Emacs

Emacs is a text editor which can be used on any operating system. This section is intended as a quick introduction and reference guide. Be warned that the complete emacs manual is approximately 600 pages (this is because emacs wants to be everything to everyone.) However, you will be able to do an incredible amount just from this section.

Emacs can be run in a text-only terminal, in which case it takes over the entire terminal. It also can run in X-Windows (under unix or linux), and under Microsoft Windows, and on a Macintosh. Below is a representative example showing emacs running in X-Windows. We will discuss how to run emacs in all three environments:

1. In a text-only terminal everything must be entered at the keyboard.
2. In a windows environment many of the common commands can be entered using the mouse and the menus (although many others still have to be entered at the keyboard).
3. In X-windows the second row contains thirteen icons of shortcuts (most of which are also contained in the menus).

For notational ease we will use the term *frame* to designate either the text-only screen or the window which is used by emacs. Also, since the term "windows" will appear so frequently in this section, we will call the "windows" version of emacs a GUI (a Graphical User Interface).

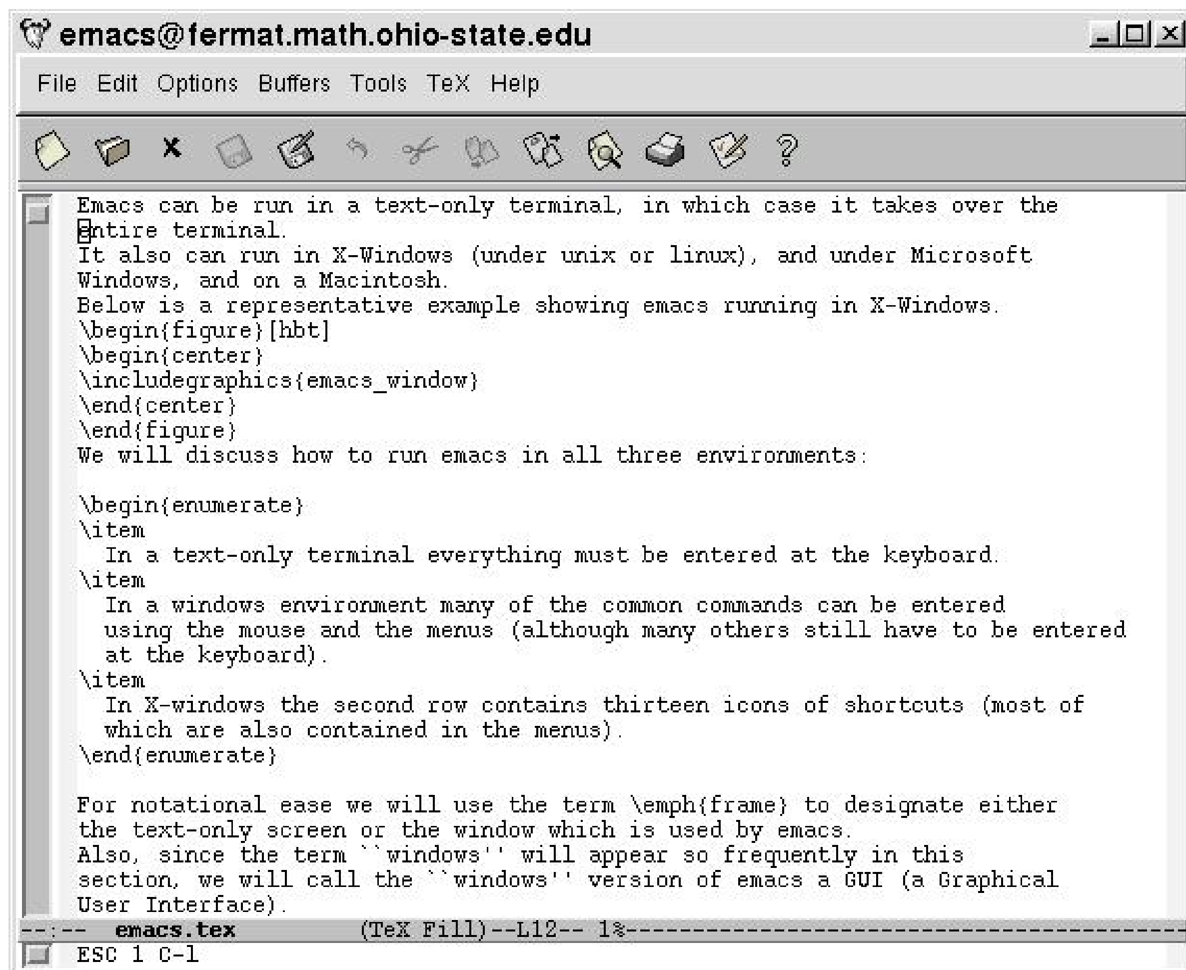
The following notation will be used in this section:

`C-` means to *hold down* the control key and type the following character. For example, `C-x` means hold down the control key and type `x`.

`M-` means to *hold down* the meta key (often called the alt key) and type the following character. Instead, you can hit the <`ESC`> key (**hit it once — do not hold it down**) and then type the following key.

There are two keys which can cause you problems, <`DEL`> and <`BACKSPACE`>(or ←):

> Keyboards almost always have both a backspace key and a delete key. Usually, but not in emacs, these keys have the same function. In emacs, <`DEL`> deletes the character to the left of the cursor while <`BACKSPACE`> is the same as `C-h` (which will appear after a few seconds at the bottom of the frame). On many keyboards, <`BACKSPACE`> is on the "main" set of keys and <`DEL`> is to the right or above the "main" set of keys. In this case, <`BACKSPACE`> is usually "aliased" to <`DEL`>(by the computer or the system administrator). Otherwise you have to do it yourself.



Text editors normally have two *distinct* modes: one for inputting text, and the other for modifying the text. In emacs these modes are not distinct: all printing characters that are typed appear in the text and all commands (i.e., which modify the text) begin with a non-printing character, either `C-` or `M-`, (or use the mouse and/or the menu). For example, if you type `k` this "k" appears in the text directly under the cursor (the blinking rectangle that indicates where the next character will be printed). If you type `C-k` the remainder of the current line is deleted. If you type "`C-x k`" (first type `C-x` and then type `k`), the file you are presently working on is erased from emacs' memory. Finally, if you type `M-k` the remainder of the current sentence is deleted.

To understand how this all appears, look at the X-window figure. The top line consists of a number of menus. (This also appears in the text-only terminal, but it is non-functional.) Below this are a number of shortcuts to common commands; this appears only in X-windows — not in Microsoft windows or on a Mac or on a text-only terminal. Below this is the window in which the text appears. Below this is a single line (called the *mode line*) in which the name of the file appears as well as other information. Finally, below this is another single line (called the *echo line*) in which the emacs commands appear (which we will discuss shortly). If a command requires further information, emacs also prompts you in this last line.

There are three very important commands to remember as you learn emacs. The first two allow you to undo many of your mistakes:

Cancel – Cancel any running or partially typed command.
⇒ ⟦keyboard: `C-g`⟧

Undo – Undo the last input or the last change (if the command is repeated, it will undo each

previous input or change in turn).
⇒ ⟦GUI: Edit → Undo⟧
⇒ ⟦shortcut: 6th icon from left⟧
⇒ ⟦keyboard: `C-_`⟧ or
⇒ ⟦keyboard: `C-x u`⟧

and the third command exits emacs:

Exit – Exit emacs.
⇒ ⟦GUI: File → Exit emacs⟧
⇒ ⟦keyboard: `C-x C-c`⟧

The exit command will ask you if you want to save each of the files that has been edited during the emacs session. If you do not save a file, then any changes made since the last time the file was saved will be lost! When asked, enter `y` if you wish to save the file, or `n` if you do not. (`n` is the correct choice only if you have made a complete mess of the editing session and want to "kill" all the changes you have made.)

Note: In a GUI you need all three mouse bottons.
The Macintosh only has one, which is the left mouse botton. To get the middle mouse button, hold down the command key (i.e., the apple) and click, while to get the right mouse hold down the options key.
Frequently on a PC the mouse only has two buttons. You similate the middle mouse button by depressing the two buttons simultaneously.

3.1.1 Starting and stopping emacs

The normal way to start the emacs editor is to type

`emacs` *file_name*

where *filename* is the name of the file you want to edit. If this file already exists, it will appear on the screen. On the other hand, if this is a new file, it will be created and the screen will be blank (because the file does not yet contain any text).

Note: You can also start emacs by simply typing `emacs` as we discuss further below.

When you work on a file in emacs, that file is stored in a *buffer* inside emacs. It is possible to work on many files at one time in emacs, in which case each file will have its own buffer, but only one buffer (and thus one file) can appear on the screen at a time by default. (But see the "Multiple Windows" subsection which describes how to have more than one buffer appear in a frame.) As you work on the file, the changes are stored in the buffer and not in the actual file on the disk. This way, you can edit the file in memory and not worry about your changes being committed to the disk until you are sure you want it to be. Keep in mind that this also means that if the computer crashes, your file editing can be lost, so it is important that your work is saved frequently — either by you or by emacs directly.

When you enter emacs, there are actually two different text areas that appear. The first contains the *current buffer* (which contains the contents of the file you are editing), and the second, called the *echo area* or *echo line*, is a single line which appears at the bottom of the frame. It can be used by emacs to give you information or to get additional information from you.

In addition, there is a *mode line* just above the echo area which describes the current state of the text area. Usually, it appears as follows.

`---:`*ch* *buffer* `(`*major and, possibly, minor mode*`)---`*line*`---`*position*`---------`

ch contains two stars, i.e., "`**`", if the buffer has been modified or "`---`" if it has not. *buffer* is the name of the text window's buffer. *major and/or minor mode* is the name of the major (and possibly a minor) mode. (See the "Mode" subsection for more information.)

Initially, the top text area only shows one buffer. However, it is possible to subdivide this area either horizontally or vertically into a number of text windows (which we will simply call *windows*). Each window can correspond to a different buffer, or they can all correspond to the same buffer. (For example, you might want to copy text from one file to another, or you might want to copy, or move, text from the part of a file to another.) There is a mode line for each window which describes its current state.

It is also possible to enter emacs without specifying a file to edit (just by typing `emacs`). (Anything you type will be stored in the buffer entitled `*scratch*`.) You can then create a new file by inputting text. However this file is not connected to any real file on the disk so it cannot be saved. In order to save this file, read the "Files" subsection below command which will be discussed below.

Warning: This is a very dangerous way to create a file. Emacs will happily let you quit without prompting you to save the file you have just created!
If you create a file in this way, make sure you save the contents almost immediately (even after typing just a single line.) If you later try to quit emacs without saving the contents, you will be asked if you want to save the file first.

3.1.2 Saving files and backup files

The changes you make to a file are stored in the buffer in memory and are only saved back to disk when:

1. you give the emacs command to save changes,
2. you exit emacs and save the changes, or
3. emacs saves the changes for you.

When you begin editing a file, emacs creates a backup copy as soon as the file is first changed. This backup copy of the original is stored in the file *filename~* (that is, the original filename followed by a "~"). If you make changes to a file and save them, you can still recover the original file after exiting emacs by moving *filename~* back to *filename*.

In addition to keeping a backup copy of your original file, emacs also saves the current buffer "every once in a while". By default it saves a backup whenever you type 300 characters or whenever you stop typing "for a while". This enables the current buffer to be recovered if emacs is aborted for any reason (for example, if you kill it by accident or the computer crashes). The changes in the file are stored in the file #*filename*# (that is, the original filename with preceding and trailing "#"). If in a subsequent session you open *filename* again, emacs will recognize the temporary file and tell you (in the echo area):

`Auto save file is newer; consider M-x recover-file.`

You can then use the command

`M-x recover-file` <ENTER>

to recover the file. (It will prompt you for the file you want to recover. Enter <ENTER> and it will ask if you want to use the auto-saved file. Type `yes` and you are all set.)

3.1.3 Files

Here are the standard commands for opening and saving files. (The ... means you will be prompted for further information in the echo area and you must end this information with a <ENTER>.)

Read a file – If the file already exists, it is opened in an emacs buffer. If the file does not already exist, a new file is created.
⇒ ⟦GUI: File → Open File ... ⟧
⇒ ⟦shortcut: 1st icon on left⟧
⇒ ⟦keyboard: `C-x C-f ...`⟧

Read a directory – If you do not remember the name of a file, or the directory it is in, the contents of a directory are shown. If you click on a file name, it is opened. If you click on a directory name, that directory is shown.
⇒ ⟦ GUI: File → Open Directory . . . ⟧
⇒ ⟦ shortcut: 2nd icon on left ⟧
⇒ ⟦ keyboard: `C-x d ...` ⟧

Save the current buffer – Save the current buffer back to disk (saves all changes made to the file).
⇒ ⟦ GUI: File → Save (current buffer) ⟧
⇒ ⟦ shortcut: 4th icon from left ⟧
⇒ ⟦ keyboard: `C-x C-s` ⟧

Save Buffer As – Save the current buffer to disk *under a new name* and also change the name of the current buffer to the new name.
⇒ ⟦ GUI: File → Save Buffer As . . . ⟧
⇒ ⟦ shortcut: 5th icon from left ⟧
⇒ ⟦ keyboard: `C-x C-w` ⟧

Insert another file – Insert the contents of another file into the current buffer at the current cursor position.
⇒ ⟦ GUI: File → Insert File . . . ⟧
⇒ ⟦ keyboard: `C-x i ...` ⟧

Note: The . . . means you will be prompted for further information in the echo area and you must end this information with a <ENTER>.

3.1.4 Moving the cursor

The terminal's cursor indicates the location where editing commands will take place. This location is called the *point*. Note that while the cursor appears to be located *at* a particular character, it is actually located immediately *before* the character. For example, if the cursor is highlighting the character "e" in "Noel, it is actually positioned before this character. If you insert the character "v", you will see "Novel" with the cursor still highlighting the "e". (Each buffer in a frame has its own point, whether or not the buffer is being displayed.)

In Windows you can move the cursor by using the left mouse button. In a text-only terminal you can use the arrow keys or

- `C-b` – Move the cursor back one character.
- `C-f` – Move the cursor forward one character.
- `C-n` – Move the cursor down to the next line.
- `C-p` – Move the cursor up to the previous line.
- `C-a` – Move the cursor to the beginning of the line.
- `C-e` – Move the cursor to the end of the line.
- `C-l` – This command does not move the cursor, but shifts the screen so the cursor is centered vertically in the window.

In Windows you scroll forward or backward by using the slider. In a text-only terminal

- `C-v` – Scroll forward one screen.
- `M-v` – Scroll backward one screen.

You can also move around in the buffer by using the following commands.

Beginning of file – Go to the beginning of the file.
⇒ ⟦ GUI: Edit → Go To → Goto Beginning of Buffer ⟧
⇒ ⟦ keyboard: `M-<` ⟧ (That is, `M-` followed by the less than character.)

End of File – Go to the end of the file.
⇒ ⟦ GUI: Edit → Go To → Goto End of Buffer ⟧
⇒ ⟦ keyboard: `M->` ⟧ (That is, `M-` followed by the greater than character.)

Go To Line – Go to the given line number.
⇒ ⟦ GUI: Edit → Go To → Goto Line . . . ⟧
⇒ ⟦ keyboard: `M-x goto-line` <ENTER> `...` ⟧ (the hyphen is part of the command)

3.1.5 Search and replace

In emacs you can search for a particular string, or a particular word, or a particular pattern either forward to the end of the buffer or backward to the beginning of the buffer. Afterwards, you can easily continue the search. To search for a particular string,

Search Forward for String – Search from point forward to the end of the buffer for the first occurrence of the string.
⇒ ⟦ GUI: Edit → Search → Search . . . ⟧
⇒ ⟦ shortcut: 10th icon from left (or 4th icon from right) ⟧
⇒ ⟦ keyboard: `C-s` <ENTER> `...` ⟧

Search Backward for String – Search from point backward to the beginning of the buffer for the first occurrence of the string.
⇒ ⟦ GUI: Edit → Search → Search Backwards . . . ⟧
⇒ ⟦ keyboard: `C-r` <ENTER> `...` ⟧

To continue searching for the same string,

Continue Searching Forward for String – Continue searching from the last found match forward to the end of the buffer
⇒ ⟦ GUI: Edit → Search → Repeat Search ⟧
⇒ ⟦ keyboard: `C-s` <ENTER> <ENTER> ⟧

Continue Searching Backward for String – Continue searching from the last found match backward to the beginning of the buffer.
⇒ ⟦ GUI: Edit → Search → Repeat Backwards ⟧
⇒ ⟦ keyboard: `C-r` <ENTER> <ENTER> ⟧

Note: If there is no further match, emacs beeps and the cursor remains at the last match.

Using keyboard input there is an easier way to initiate a search and to continue searching. You can initiate an incremental search which begins as soon as you type the first character in the search string. As you keep typing, emacs continues searching for the entire string you are typing in. For example, if you type "find" it first finds the next "f", then the next "fi", then the next "fin", and finally the word "find" you are searching for. If then you use <DEL> to delete the "d" in "find" the cursor will move back to the occurence of "fin" it previously found (which might be, for example, in the word "final").

`C-s ...` – Incremental search forward.

`C-r ...` – Incremental search backward.

To continue a search using the current string, there are two possibilities. If you *have not* done anything since the last incremental search forward or backward, then a simple `C-s` or `C-r`, respectively, continues the search. If you *have* done something else then

`C-s C-s` – Continue searching forward.

`C-r C-r` – Continue searching backward.

You can also search for a particular word or group of words without regard for how the words are separated. For example, if you want to search for the words "keep typing" (which appeared previously) you don't care whether there is one space or two spaces or . . . between the two words; in addition, you don't care whether the word "keep" ends a line and the word "typing" begins the next line. Often you don't even care if there are punctuation marks between the words. For example, the "keep" and "Typing" in

> Until tomorrow, can't he keep? Typing away as I am to try to finish this site guide, I simply don't have time to aspirate the asp tonight."

will be found by the word search command.

`C-s` <ENTER> `C-w ...` – Search forward for the word or words ignoring spaces and/or punctuation characters.

`C-r` <ENTER> `C-w ...` – Search backward for the word or words ignoring spaces and/or punctuation characters.

There is a more advanced type of search that we will just mention. A *regular expression* (*regexp*, for short) is a "wildcard" pattern. That is, certain characters have special meanings that allow you to search for a general pattern rather than a particular character. For example, "x[0-9]" stands for the letter "x" followed by any digit. If you need to use this, read the manual.

Search Forward for a *regexp* – Search forward using a pattern given by a regular expression.
⇒ ⟦ GUI: Edit → Search → Advanced Search/Replace → Search Regexp . . . ⟧
⇒ ⟦ keyboard: `C-M-s ...` ⟧

Search Backward for a *regexp* – Search backwards using a pattern given by a regular expression.
⇒ ⟦ GUI: Edit → Search → Advanced Search/Replace → Search Regexp Backwards . . . ⟧
⇒ ⟦ keyboard: `C-M-r ...` ⟧

There are two different commands for replacing strings. The first is completely straightforward, unlike the second, but it is not as safe. To do an unconditional replacement

`M-x replace-string` <ENTER> `...` – Replace every occurence of the *old_string* with the *new_string* from the point to the end of the buffer.

Note: You will be prompted for the *old_string* which you will end with a <ENTER> and then for the *new_string* which you will again end with a <ENTER>.

If you only want to replace some strings, or you want to make sure that you have entered the two strings correctly,

Query Replace – Emacs queries you whether or not you want to do each replacement.
⇒ ⟦ GUI: Edit → Search → Replace . . . ⟧
⇒ ⟦ keyboard: `M-% ...` ⟧

Note: You will be prompted for the *old_string* which you will end with a <ENTER> and then for the *new_string* which you will again end with a <ENTER>.

When queried, there are a number of replies you can give. The most common are

<SPACE> Yes, replace this string and continue to the next occurence.

<DEL> No, do not replace this string but continue to the next occurence.

<ESC> No, do not replace this string and quit.

, Yes, replace this string and display the result for the user to check. At this point both <SPACE> and <DEL> continue to the next occurence. `C-_` and `C-x u` both undo this last replacement and **quit**.

`!` replace this string and all following occurences without asking again.

`C-h` display a message listing all the options. Then enter whatever you want to do.

Searches in emacs ignore the case of the text they are searching for if you specify the text completely in lower case. Thus, if you are searching for "you", emacs will also find "You" and even "YOU". However, if any character is uppercase, then the search is case sensitive. Thus, if you are searching for "You", you will not find "you" or "YOU".

A similar situation holds for replacement. If the *old_string* is completely lower case, then the search will be case insensitive. However, the replacement will not be. For example,

```
M-x replace-string <ENTER>you<ENTER>me
```

will replace "you" with "me", "You" with "Me", and "YOU" with "ME". On the other hand, if any character in *old_string* is upper case, then *new string* is inserted exactly as typed.

There are times when you have run a command which requires you to enter some text into the echo area and then you (or emacs) have quit the command. To rerun this last command with the *same text*, enter

```
C-x <ESC><ESC><ENTER>
```

and you will see the actual Lisp command corresponding to what you entered.[1]

3.1.6 Deletions and cut and paste

It is easy to delete one character at a time using keyboard input.

`<DEL>` – Delete the last character (i.e., to the left of the cursor).

`C-d` – Delete the present character (i.e., under the cursor).

Note that the `<ENTER>` at the end of each line is a character and can be deleted by any deletion command. This is the normal way to join to lines together into one line.

It is also easy to delete larger amounts of text at one time (called *killing* rather than *deleting*). When this is done the material killed is saved for later use (in the *kill buffer*) and can be inserted anywhere else in the text (of any window) by moving the cursor to the desired point. Simply set the *mark* at one end of the text and the point at the other. The text between the point and the mark is called the *region*.

In Windows just hold down the left mouse button and move it from one end of the region to the other. Alternately, click the left mouse button at one end of the region to set the point and click the right mouse button at the other end (which sets the mark at the other end and the point at the current end). If you continue clicking the right mouse button, the region continues to be set between the mark at the other end and the current point.

Using keyboard input you set the region by

`C-<SPACE>` Set the mark at the current position.

`C-@` Also sets the mark at the current position.

`C-x C-x` Switch the mark and the point (to determine the bounds of the region since the curson can only highlight one end of it).

Next you can either delete the region or copy it to the kill buffer (which saves the last 16 items of text killed and all of these are accessible). You can also reinsert it (this is called "yanking").

Delete the Region – Delete the region and put it in the kill buffer.
⇒ ⟦GUI: Edit → Cut⟧
⇒ ⟦shortcut: 7th icon from left (or middle icon)⟧
⇒ ⟦keyboard: `C-w`⟧

Copy the Region – Copy the region and put it in the kill buffer.
⇒ ⟦GUI: Edit → Copy⟧
⇒ ⟦shortcut: 8th icon from left (or 6th icon from right)⟧
⇒ ⟦keyboard: `M-w`⟧

[1] Emacs is actually written in Lisp, which is an interpreted language. When you enter a command, it is translated to Lisp code and then executed. If you know Lisp, you can modify emacs to your hearts content!

Reinsert the last region killed – Reinsert (paste) last killed text (i.e., put it back in the text at the present cursor location).
⇒ ⟦GUI: Edit → Paste⟧
⇒ ⟦shortcut: 9th icon from left (or 5th icon from right)⟧
⇒ ⟦keyboard: `C-y`⟧ (This is called *yanking* in emacs.)

Reinsert a previously killed region – Reinsert (paste) a region which was previously killed.
⇒ ⟦GUI: Edit → Select and Paste⟧ You will then be shown all the regions in the kill buffer.
⇒ ⟦keyboard: `M-y`⟧ This must follow *immediately* the last `C-y` or another M-y and replaces the text just previously yanked with the preceding item in the kill buffer. To replace the second-from-the-last item type `C-y M-y`. To replace the third-from-the-last item type `C-y M-y M-y`, etc.

Mark entire buffer – Make the entire buffer a region.
⇒ ⟦GUI: Edit → Select All⟧
⇒ ⟦keyboard: `C-x h`⟧

In keyboard input there are a few commands that kill a specified amount of text. It is important to realize that each kill command usually pushes a new item onto the kill buffer. *However*, if two or more of these commands occur in a row (i.e., with *no* keystrokes in between) they combine their text into a single item in the kill ring. For example, if the cursor is at the beginning of a paragraph that is three lines long, then "`C-k C-k C-k C-k C-k C-k`", i.e., 6 `C-k`'s in a row, will kill the entire paragraph and put it into the last item in the kill buffer. (Recall that the first `C-k` kills the line of text and the second `C-k` kills the <`ENTER`> so it takes *two* `C-k`'s to kill an entire line of text.

`C-k` – Kill from the cursor to the end of the line. However, if the cursor is already at the end of the line it only kills the <`ENTER`> at the end of the line.

`M-d` – Kill the next word.

`M-\` – Kill all the whitespace surrounding the point.

`M-z` *char* – Kill all the text to the next occurence of the character *char*.

It is also important to realize that there is only *one* kill buffer. As we will discuss in the next subsection you can kill (or copy) text in one buffer and reinsert (or yank) it into another buffer. This is the standard way to move (or copy) text from one file to another.

3.1.7 Multiple windows

When you first run emacs you open one window and can only work on one file at a time. However, it is possible to work on two parts of the same file (for example to move — or copy — text from one part to the other) or to work on two different files at the same time. Recall that the text you are editing in emacs resides in an object called a buffer (and that the changes you make are made to the buffer and not to the file on disk). Each time you visit a file, either by entering emacs with a specified file (i.e., typing "`emacs` *filename*") or by opening a new file, a new buffer is created to hold the file's text. At any time, one and only one buffer is selected (called the *current buffer*). The cursor is always connected to the current buffer. If there are two or more buffers in use at the same time you can easily change the current buffer.

You can easily split one window into two, either horizontally or vertically — each of which will show the same text. You can move the cursor in each window to show a different part of the same buffer; then any change in one window changes the contents of the buffer and so the change also applies to the other window. (This makes it easy to copy or move text from one part of a file to another.) However, if each window is showing a different buffer then each window is completely independent. Text can still be copied or moved from one buffer to the other by killing and yanking. To do this you would split the screen into two windows and then either load another file in one of the windows or change the buffer in this window. You can then move between the buffers. (The window containing the cursor is called the *current* window.)

Split Window Vertically – Split the current window into two, one above the other.
⇒ ⟦GUI: File → Split Window⟧
⇒ ⟦keyboard: `C-x 2`⟧

Split Window Horizontally – Split the current window into two, one beside the other.
⇒ ⟦keyboard: `C-x 3`⟧

Delete Windows – Delete all the windows, except the current one. (The buffers are not deleted, but they are no longer visible.)
⇒ ⟦GUI: File → Unsplit Windows⟧
⇒ ⟦keyboard: `C-x 1`⟧

Delete Current Window – Delete just the current window. (The other window(s) grow in size.)
⇒ ⟦keyboard: `C-x 0`⟧ (zero, not "oh")

Select Other Window – Move the cursor to another window (i.e., make a different window the current window).
⇒ ⟦GUI: click the left mouse button in the window's mode line⟧
⇒ ⟦keyboard: `C-x o`⟧ ("oh", not zero)

Switching Buffer in Window – Change the buffer which appears in the current window.
⇒ ⟦GUI: Buffers⟧ Simply click on the desired buffer.
⇒ ⟦keyboard: `C-x b ...`⟧ Switch to a different buffer.
⇒ ⟦keyboard: `C-x C-b`⟧ Find out the names of all the currently existing buffers (if you don't remember the name of the buffer you want to switch to).

Finally, there are times when you want to kill a buffer. Note that the disk file is not killed but it is no longer in a buffer in emacs. If you kill a buffer without saving it, then any and all changes made to the buffer will be lost. However, suppose you have created a large number of buffers (possibly because you were looking through a number of files for something) that you no longer need. Then killing the buffer(s) makes good sense. Alternatively, suppose you have made all the changes you want to a buffer and have saved those changes back to the corresponding file. Then you might want to kill the buffer simply to tidy up your workspace. (Emacs will query you if you try to kill a buffer whose changes have not been saved.)

Kill Current Buffer – Kill the current buffer. (Emacs will query you before actually killing the buffer if there are unsaved changes.)
⇒ ⟦shortcut: 3rd icon from left⟧
⇒ ⟦keyboard: `C-x k`⟧

3.1.8 Odds and ends

To convert an entire region to upper or lower case

`C-x C-l` – Convert region to lower case.

`C-x C-u` – Convert region to upper case.

In section 1.11 we discussed the different end of line characters for UNIX, Microsoft Windows, and Macintosh. You can always tell which format is being used on the mode line. If the format is correct for the operating system you are currently on, you will see

```
--:
```

at the beginning of the mode line. However, if the format is different you will see

`--(DOS)` or `--(UNIX)` or `--(MAC)`

Emacs will happily use any of these formats and even save the file using this format. However, if desired, you can change the format when you save the file.

Change the end of line character –
⇒ ⟦ GUI: Options → MULE → Set Coding Systems → For Saving this Buffer ⟧
and then in the mode line you will enter:
`dos` or
`unix` or
`mac`
If you cannot remember what to do, enter `?` and you will see *all* your options
⇒ ⟦ keyboard: `C-x <ENTER> f` ⟧

3.1.9 Modes

Emacs has *major modes* for various programming languages. These help you code your programs to make it easier to get the syntax correct:

> "A programming language major mode typically specifies the syntax of expressions, the customary rules for indentation, how to do syntax highlighting for the language, and how to find the beginning of a function definition. It often customizes or provides facilities for compiling and debugging programs as well."

For example, emacs will happily indent your programs for you. It is amazing how often this will find simple errors. For example, in Fortran 77 if you enter

```
      sum1 = 0.
      sum2 = 0.
      do 20 i = 1,n
         sum1 = sum1 + i**3
         sum2 = sum2 + i**4
10       continue
```

you will immediately see that the indentation for the `continue` statement is wrong — well, at least it looks wrong. The problem is that the label should be `20` and not `10`. Emacs recognizes that the `continue` statement is incorrect and so it does not change the indentation.

Emacs will also handle Fortran 90, which uses a free format. For example, if you enter

```
sum1 = 0.
sum2 = 0.
do i = 1,n
   sum1 = sum1 + i**3
   sum2 = sum2 + i**4
   end <TAB>
```

the last line will be converted to

```
end do
```

after the "do" which begins the loop is momentarily highlighted.

In C, a common mistake is

```
main() {
  int i, sum1 = 0, sum2 = 0;
  su
  for ( i = 1; i <= 10; i++ )
    sum1 = sum1 + pow(i,3);
  sum2 = sum2 + pow(i,4);
```

Here the problem is that the user forgot to use braces ({. . . }) for the block of the `for` statement. Since the `sum2` line is not indented, it is clear that the `for` loop is only being applied to the statement which immediately follows.

One very important feature of emacs is that it shows automatically how parentheses (and any other matching delimiters) match in the text. When you type a closing delimiter, the cursor moves momentarily to the corresponding opening delimiter if it is visible on the screen. If not, it displays the line in which the opening delimiter appears in the echo area. If the delimiters do not match, an warning message appears in the echo area.

To indent a line or a number of lines,

`<TAB>` – Indent a single line.

`C-M-\` – Indent a region. This is often useful if you have copied a number of lines from somewhere else and the indentation is no longer correct.

It is also easy to move around in a code. Most (if not all) program languages can be split into smaller program units. These are usually called functions or subroutines or macros or subprograms; in emacs these are all subsumed under the term *defuns*.

`C-M-a` – Move up to the beginning of the current defun.

`C-M-e` – Move down to the end of the current defun.

`C-M-h` – Make the entire defun a region.

Emacs determines which major mode to use by looking at the file name extension. For example, "f" refers to Fortran 77 or Fortran 90, "c" refers to C, "cpp" refers to C++, etc. The GUI adds an additional menu item for each language. Consult it for commands which are specific to that language. If you are using an extension that emacs does not recognize, you can change to the major mode you desire by

`M-x` *language*`-mode <ENTER>` – *language* includes Fortran ("fortran" or "f90"), C ("c"), C++ ("c++), as well as many others.

3.1.10 Compiling a program

You can compile a program in emacs and easily look at the source lines where your errors occur.

Compile – Run a compiler in emacs.
⇒ ⟦GUI: Tools → Compile ... ⟧
⇒ ⟦keyboard: `M-x compile <ENTER> ...`⟧

Emacs assumes you want to compile the program by using a make file and enters "`make -k`" in the minibuffer. If you complete this command, the program requested will be compiled and linked. However, you can delete this command by typing `<DEL>` repeatedly and then typing in a compiler command such as "`f77 -g` *filename*". Note that emacs will first ask you if you want to save each file that has been changed. If you do not save a file then the compilation will occur on the unchanged file (since the file on disk and *not* the buffer in emacs is being compiled)!

Emacs adds a new window containing the `*compilation*` buffer which lists any and all error messages. You can go through these errors one after the other or find the source line for a particular error message.

Next compiler error – Find the line of the source code which contains each error in turn.
⇒ ⟦keyboard: `` C-x ` ``⟧ (**not** the apostrophe but the grave symbol, i.e., the backward apostrophe)

Particular compiler error – Find the line of the source code which contains this particular error.
⇒ ⟦GUI: middle mouse button⟧ click the middle mouse button on the error line.
⇒ ⟦keyboard: `<ENTER>`⟧ hit `<ENTER>` while the cursor is on the error line.

When all the errors have (hopefully) been corrected, compile the program again and see what happens.

3.1.11 Non-printing characters

In emacs it is easy to input most characters and to see most characters on the screen. One character that is a problem is <TAB>, the TAB character. It can be input by hitting the TAB key. However, the <TAB> character cannot be seen on the screen since it appears to be just one or more spaces. Since this character is essential when using the program `make` (see Section 4.1) it is important to know whether this character is really in a file. There is one easy way to find <TAB> — just search for it by searching for the <TAB> character (using the menu or typing "`C-s <TAB>`". This will find all the tabs in the file. There are emacs commands for putting in and taking out tabs from a file:

`M-x tabify` <ENTER> – *Inside the current region* replace, whenever possible, spaces with tabs.

`M-x untabify` <ENTER> – *Inside the current region* replace all tabs with spaces.

Occasionally, it may be necessary to put a control character into a file (as opposed to having it be an emacs command). This is done by

`C-q` *char* – *char* is put into the file where *char* can be *any* single character, including any control character. For example, "`C-q C-c`" puts the character `C-c` into the file.

3.2 Vi and Vim

The vi (short for "visual") editor is another text editor available on our UNIX systems. Unlike emacs, vi separates its operation into two modes:

- *Command Mode* is used to enter commands for the vi editor (such as save file, find word, etc)
- *Edit Mode* is used to edit the text of the file

While this may seem awkward at first, it allows commands to be executed without multiple keystrokes such as the control commands that emacs uses.

Vi is somewhat complex to use because edit mode did not allow you to do much at all, not even move the cursor around. Vim (vi improved) was created to address some of these issues. The most notable difference between vi and vim is that you can use the arrow keys to navigate while in edit mode and that labels at the bottom of the window tell if you are in edit mode and list the current line and column number. Both vi and vim are installed on our system, vim is capable of much more than vi, and we recommend using it instead. However, most of the advanced features that vim offers over vi are out of the scope of this document. More information can be found at `http://www.vim.org`. This section will focus on vim. However the commands for both are the same.

Many of the commands used function relative to the cursor position (for example, delete the caracter under the cursor). The cursor in vim is marked on the screen as a solid white rectangle hi-lighting a character.

3.2.1 Starting and stopping vim

Vim is normally started with the command

`vim` *filename*

where *filename* is the name of the file you want to work on. If the file already exists, it will be opened for editing, otherwise it will be created as a new blank file. When you start vim, you will be in command mode. The bottom line on the screen is used to display information on the file you are editing. One the left side, you will see the name of the file you are editing. This area is also used to display commands you enter, prompt you for information, or display status messages. Toward the right side of the screen, you will see two numbers separated by a comma. The first is the row the curor is on in the file and the second is the column the cursor is on in the file. On the far right, a number will indicate what percentage of the document you are at.

3.2.2 Moving the cursor

In command mode, there are several command you can use to move the cursor around (besides the arrow keys). Typing any of the following commands in command mode will move the cursor to the indicated position.

- `nG` moves cursor to line *n* (or, just `G` alone will go to the end of the file)
- `0` (zero, not letter o) move to the beginning of theline
- `$` move to the end of the line
- `^` move to the first non-whitespace character on the line
- `+` move to the first character on the next line
- `-` move to the first character on the previous line
- `C-f` scroll forward one screen
- `C-b` scroll back one screen
- `C-e` show one more line at the bottom of the screen
- `C-y` show one more line at the top of the screen
- `L` go to the bottom of the screen
- `fchar` forward to the character *char* on the line
- `Fchar` backward to the previous character *char* on the line
- `w` move to the next word (stops at puntuation)
- `W` move to the next word (skips punctuation)
- `b` move to the previous word (stops at punctuation)
- `B` move to the previous word (skips punctuation)
- `e` go to the end of the word (excluding punctuation)
- `E` go to the end of the word (including punctuation)
- `)` go to the next sentence
- `(` go to the previous sentence
- `}` go to the next paragraph
- `{` go to the previous paragraph

Don't let this scare you! You can still navigate files just using the arrow keys, these commands are just a convenience to navigate around your files. Quick references for vi commands are available in MW430.

3.2.3 Enter edit mode

To edit text, you must first enter edit mode. There are several commands to do this, each enters edit mode, the difference is where the cursor is placed for editing relative to the current cursor position. The following commands will switch to edit mode, to enter the command, simply type the command.

`a` append text after the cursor

`A` append text at the end of the current line

`i` insert text before the cursor

`I` insert text at the beginning of the line

`o` (letter o, not zero) insert text on the line below the cursor

`O` (letter O, not zero) insert text on the line above the cursor

3.2.4 Exiting edit mode

To exit edit mode and go back to command mode, hit the escape key.

3.2.5 Working with text

There are several commands you can use to delete, cut, copy, and paste text in vim.

`dd` cut/delete the current line

`D` cut/delete to the end of the current line

`x` cut/delete the character under the cursor

`X` cut/delete the character before the cursor

`y` yank (copy) text

`Y` yank (copy) current line

`p` paste text after the cursor

`P` paste text before the cursor

Copying and cutting text first requires the desired text to be selected. If you want to cut or copy the current line, you can just use dd or Y respectively. Otherwise, you will first need to select the text to cut or copy. To do this, you will first need to enter visual mode by entering the command `v`. Once in visual mode, you can navigate around the file using any of the normal navigation commands or arrow keys, and you will see the text become hilighted. When you are satisfied with your selection, you can then use the `x` or `y` commands to cut or copy the selected text. It will be stored in a buffer and can then be pasted into the file using the `p` command.

3.2.6 Searching and replacing text

The following commands can be used to search and replace text in vim.

/*string* search forward for the text *string*

?*string* search backward for the text

`n` repeat search forward

`N` repeat search backward

`:s/`*old_string*`/`*new_string*	replaces the next occurence of the text *old_string* with the text *new_string*
`:s/`*old_string*`/`*new_string*`/g`	replaces all occurences of the text *old_string* with the text *new_string* on the current line
`:%s/`*old_string*`/`*new_string*`/g`	replaces all occurences of the text *old_string* with the text *new_string* anywhere in the file

For example, if you wanted to find the text "I love vi!" in the file, you would use the command

```
/I love vi!
```

If you wanted to replace all occurences of the text "I love emacs!" with "I love vi!" in the file, you would use the command

```
:%s/I love emacs!/I love vi!
```

3.2.7 Miscellaneous commands

Here are a few more commands you may find useful in vim.

`.`	repeat last command
`U`	undo changes on current line
`u`	undo last command
`J`	join end of current line with the next line
`:r`*file*	inserts text from the file *file*
`C-g`	shows the current status (including the filename, if the file has been modified, the line you are currently on, and the fraction of the distance you are in the file)

Chapter 4

Odds and Ends

4.1 Make

`Make` is a program that is used to sort out dependency relations among files in a software project. Rather than trying to explain the program in its full generality we will concentrate on one specific example. Afterwards we will briefly consider a few other examples to show the range of uses of this program

Suppose your main program is contained in `testit.f`, you have subroutines contained in two files `testsub1.f` and `testsub2.f`, and you want to include the double precision version of the Linpack library to solve a matrix equation. You can generate the executable code, `test.e`, in one command by

```
f77 -g -o testit.e testit.f testsub1.f testsub2.f -llinpackd
```

(See section 2.1 for a discussion of how to compile and load a program, and how to use libraries.)

Suppose later you make a change in the file `testsub1.f`. Then you could re-execute the above command to generate a new executable code. However, this might take a long time. In addition, it might take a few tries to get all the Fortran bugs out of the code. Thus, it would be preferable to generate object files for each source file and then link these object files together.

```
f77 -g -c testit.f
f77 -g -c testsub1.f
f77 -g -c testsub2.f
f77 -g -o testit.e testit.o testsub1.o testsub2.o -llinpackd
```

If `testsub1.f` is later changed, the commands necessary to generate a new executable code would be

```
f77 -g -c testsub1.f
f77 -g -o testit.e testit.o testsub1.o testsub2.o -llinpackd
```

With these commands the source code can be debugged first, and then all the object codes can be linked together.

However, there are a number of concerns with this method of generating a new executable code:

- You might forget to recompile the changed source code.
- You might forget to relink all the object files to obtain a new executable code.
- You might recompile the *wrong* source code (say `testsub2.f`).
- And, what might be even worse, you might worry every time that your code does not work "correctly" that you forgot to recompile or relink something.

By using `make`, you can avoid all these worries. The program `make` checks (among other things) whether the executable code is up-to-date. If it is not, it recompiles the necessary source files and generates a new executable code. In addition, it will do this inside emacs so that you can compile and link the necessary files even as you are modifying one or more source files!

In order to do all this, `make` must know which files depend on which other files. These dependencies are listed in the *description file* that must be named either `Makefile` or `makefile` and must reside in the same directory as the source files. The "makefile" (let's call it "Makefile") could be the following (where the line numbers on the right are not part of the source file but are used in describing this file below).

```
# Example Makefile for testit                                    [[1]]
LIBS = -llinpackd                                                [[2]]
test.o:         testit.f                                         [[3]]
                f77 -g -c testit.f                               [[4]]
testsub1.o:     testsub1.f                                       [[5]]
                f77 -g -c testsub1.f                             [[6]]
testsub2.o:     testsub2.f                                       [[7]]
                f77 -g -c testsub2.f                             [[8]]
testit.e:       testit.o testsub1.o testsub2.o                   [[9]]
                f77 -g -o testit.e testit.o testsub1.o testsub2.o ${LIBS}  [[10]]
```

Warning: Before discussing this makefile, there is one very important point which cannot be stressed too highly. **THIS MAKEFILE CONTAINS NON-PRINTING CHARACTERS YOU CANNOT SEE BUT WHICH ARE ESSENTIAL.**

The listing below is the same as above but the non-printing characters are shown.

```
# Example Makefile for testit                                     [[1]]
LIBS = -llinpackd                                                 [[2]]
test.o:<TAB><TAB>testit.f                                         [[3]]
<TAB><TAB>f77 -g -c testit.f                                      [[4]]
testsub1.o:<TAB>testsub1.f                                        [[5]]
<TAB><TAB>f77 -g -c testsub1.f                                    [[6]]
testsub2.o:<TAB>testsub2.f                                        [[7]]
<TAB><TAB>f77 -g -c testsub2.f                                    [[8]]
testit.e:<TAB><TAB>testit.o testsub1.o testsub2.o                 [[9]]
<TAB><TAB>f77 -g -o testit.e testit.o testsub1.o testsub2.o ${LIBS}   [[10]]
```

The tab characters are not necessary in lines 3, 5, 7, or 9 (which are called the "dependency" lines, as discussed below). However, a tab character **MUST** begin lines 4, 6, 8, and 10 (which are called the "command" lines, as discussed below). That is how `make` differentiates dependency lines from command lines. The remaining tab characters are extraneous from the program's point of view, but they improve the readability of the program as can be seen by comparing the two listings.

Now let us discuss how this makefile works. Line 1 begins with a `#` and designates a comment. This line is not necessary but is included to show how to put in a comment line. Everything following `#` on a line is ignored by `make`. To improve readability lines can also be completely blank.

Line 2 sets the variable `LIBS` equal to "`-llinpackd`". The first word in this line is the variable and everything following the equal sign on the line is the value (or contents) of the variable. This variable is referenced later in line 10 where the variable is replaced by its contents. (Note how the variable must be preceded by a `$` and surrounded by braces.) This variable is not essential because the linking command could include "`-llinpackd`" directly, rather than through this variable. However, there are a number of reasons why it might be preferable to use a variable.

- There may be a number of programs in the makefile which require this library and this way it need only be written once.
- One library may require the one or more other libraries also be linked. This is again easier to do once (especially since you might forget to add the additional library or libraries if there are many programs in the makefile).

- This makefile might be used on two different computers and the names of the libraries might be different on the two machines. This can be easily handled by commenting out the line containing the other computer's library name.

Lines 3–10 in this makefile contain the dependency information which the program needs to determine if the executable code is up-to-date. Each lines consists of either a *dependency line* (lines 3, 5, 7, and 9) or a *command line* (lines 4, 6, 8, and 10). To the left of the colon on a dependency line is one or more *targets*. To the right of each colon are the *component* or *components* on which each target depends. Each command line **MUST** begin with <TAB> as discussed above and contains the necessary information to generate a new target when needed.

Let us first focus on lines 3 and 4. The date and time when the target file, `testit.o` (i.e, the object file), was last modified is compared with the date and time that the *component* file, `testit.f` (i.e., the source file), was last modified. If the object file is *newer* than the source file, then it is assumed that everything the current source file generated the current object file. However, if the object file is *older* than the source file, then it is assumed that the source file was modified *after* the object file was generated and so a new object file should be generated by executing any and all command lines following the dependency line (until the next dependency line occurs. The command line, line 4, gives the compilation command that must be executed in order to generate a new object file.

Lines 5 and 6 are similar and so are lines 7 and 8. The dependency line for the executable code in line 9 is more complicated. The target file, `testit.e` (the executable code), has as its *component* files the object files `testit.o`, `testsub1.o`, and `testsub2.o`. The dates and times of these files are not immediately checked. The program first checks if the *component* files are themselves target files and finds that they are on lines 3, 5, and 7. (It does not matter whether these target lines precede or follow the line on when they are *component* files. The program checks the entire file.) These dependencies are first checked as described above. Only then are the dates and times of the executable code and the object codes compared. If any or all of the object codes are *newer* than the executable code then the command line, line 10, is executed and a new executable code is generated.

To check if the executable code, `testit.e`, is up-to-date simply type

```
make testit.e
```

Any and all of the source codes that had been modified would be recompiled and, if there were no Fortran bugs, the executable code would then be generated. If you simply wanted to re-compile one particular source code (say `testsub1.f`) you would type

```
make testsub1.o
```

(Note that `make` can take as its argument *any* dependency file.) But remember that `testsub1.f` will not be recompiled if `testsub1.o` is newer. In this case you will get the message

```
make: 'testsub1.o' is up to date
```

If you make a mistake and type in an incorrect argument, such as

```
make testit
```

you will get the message

```
make: Don't know how to make testit. stop
```

This means that the argument is not a dependency file.

As mentioned at the beginning of this section, `make` can sort out dependency relations among all the files in a software project, which may encompass dozens or even hundreds of files in many different directories. The above example, however, should give you a good idea of the basic features of `make`. In the remainder of this section we will amplify some of these basic features. Let us emphasize a few points here.

- There is only <u>*one*</u> makefile for each directory so all the programs must be contained in this file.
- The order in which the programs are listed is immaterial and the dependency lines can be put in any order (although they must be followed by the proper command lines).

- There can be more than one command line for each dependency line (as we will show in the library example below.

Note that even if a program (say with source code `simple.f`) is contained completely in one file with no libraries it is useful to use a makefile. If the source code is compiled in emacs then, after typing `M-x compile` <ENTER> (or using the menu), emacs responds with "`make -k`". If the dependencies for `simple.f` are contained in the makefile you can merely type `simple.e`. However, if you are not using a makefile you will have to delete emacs' response and type "`f77 simple.f`" or "`f77 -g simple.f`" or even "`f77 -g -o simple.e simple.f`". Which seems simpler?

Finally, here is a makefile which generates a library. Assume that there are eight files (`subfile1.f`, ..., `subfile8.f`) which make up the library file `subfile.a`. Suppose you are the user `zola`. You want to generate the library file, and then move it to your subdirectory `~/lib` (see Figure 1.1). The following makefile does all this.

```
COMPILE = f77
COPTN   = -g
subfile1.o:     subfile1.f
                ${COMPILE} ${COPTN} subfile1.f
subfile2.o:     subfile2.f
                ${COMPILE} ${COPTN} subfile2.f
subfile3.o:     subfile3.f
                ${COMPILE} ${COPTN} subfile3.f
subfile4.o:     subfile4.f
                ${COMPILE} ${COPTN} subfile4.f
subfile5.o:     subfile5.f
                ${COMPILE} ${COPTN} subfile5.f
subfile6.o:     subfile6.f
                ${COMPILE} ${COPTN} subfile6.f
subfile7.o:     subfile7.f
                ${COMPILE} ${COPTN} subfile7.f
subfile8.o:     subfile8.f
                ${COMPILE} ${COPTN} subfile8.f
subfile.a:      subfile1.o subfile2.o subfile3.o subfile4.o subfile5.o subfile6.o \
                        subfile7.o subfile8.o
                ar rcv subfile.a subfile1.o subfile2.o subfile3.o subfile4.o \
                        subfile5.o subfile6.o subfile7.o subfile8.o
                mv subfile.a ${HOME}/lib/
```

There are three new wrinkles here.

1. The first is how to continue a line. This is done by ending the line with a backslash (i.e., a "\"). (The following line is indented just for ease in reading.)
   Warning: the backslash must be the last character on the line; that is, there cannot be any spaces following it.
2. The second is that there are two command lines following a dependency line. The first line (i.e., `ar rcv ...`) generates a new library file and the second (i.e., `mv ...`) moves it to the directory `~/lib` .
3. The third is that the variable `HOME` is the enviroment variable (which we discussed in section 1.9.2) which contains your home directory. It is used in the last line to put the file `subfile.a` into the directory `~/lib` .

4.2 Debugging Your Programs

4.2.1 What is a debugger?

A *debugger* is an application (i.e., a computer program) that runs your computer program. When you run your program you type

program_name

and your program begins executing. If your program runs correctly, fine. If it does not, you would like to find out why. This is the purpose of a debugger.

When you use a debugger you type

debugger program_name

where, for this discussion, *debugger* can be `dbx`, `gdb`, `ldb`, or `ddd`. At this point your program is **not** running. Instead, the debugger is running and waiting for instructions from you. You have to tell the debugger what it should do to your program, i.e., *program_name*. You have many options at this point:

- You can step through your source code executing one line at a time and inspect, or even change, the value of any variable.
- If your program is crashing when you run it normally, the debugger can usually tell you where it crashed and, with a little effort, why.
- If an `if` statement is not behaving as expected, you can see what is happening each time the statement is executed and figure out what is going wrong.
- If your program goes into an infinite loop, you can stop it and find out why.
- If a variable in a subprogram seems to be screwed up, you can trace it all the way back to its declaration.

You actually have many more options available to you — but these are probably the most common.

When you learn a programming language, you are generally taught to debug your programs by putting in lots of `cout` statements or `print` statements or `printf` statements or `system.out.println` statements or `write` statements. This allows you to view the values of variables at certain strategic points and also to view the order of execution of your code.

Printing out everything in sight is certainly the easiest way to debug your program in the sense that you do not have to learn anything new. However, it is frequently not the fastest way to debug your program. In fact, if your program is reasonably long, it may be an incredibly slow way to debug your program. The reason it is used by so many people is that they do not want to spend the time to learn how to use a debugger.

That is the purpose of this chapter: it does not take much time or effort to learn how to use a debugger — and with a GUI (Graphical User Interface) it can be positively fast and easy.

There are many debuggers out there, since, generally, every compiler has its own debugger. The ones we will discuss are `dbx`, which is the "standard" UNIX debugger, and `gdb`, which is the "standard" GNU debugger for C and C++. Neither of these work very well with fortran. For Intel compilers there is `ldb`, which is basically a subset of `gdb`, so there is nothing new to learn. For C++ it is probably easier just to stick with GNU's compiler. However, `ldb` works well with fortran, unlike `dbx` or `gdb`. (See Subsection 2.1 for more details about the available compilers.)

In this document we will concentrate on `gdb` and explain in some detail its "basic" commands. Then we will discuss `ddd`, a GUI front end for `gdb`. Later, we will briefly summarize the `dbx` commands and the `ldb` commands. We will not actually show *how* to use `gdb` — there are many tutorials on the web which do that — but only discuss the most common commands.

4.2.2 Basic gdb commands

Again, the program you want to run is your program, i.e., *program_name*. Instead, you run the debugger, which in this subsection is `gdb`, and you instruct it how to run your program . You run `gdb` by typing

`gdb` *program_name*

or by typing

`gdb`

and you will later have to use the `file` command to indicate which file to execute.

Warning: **First, you have to compile your program with the `-g` option. See Subsection 2.1.1 for more details.**

Now that you are running `gdb`, you instruct it to run your program by using the `run` command. Usually you will want to set *breakpoints* beforehand. A breakpoint is used to stop your program when a particular line of the source code is reached.

If you begin running your program without setting any breakpoints or watchpoints, your program will run as if you had run it directly (i.e., not inside `gdb`). There are a number of possibilities:

- Your program will abort.
- Your program will go into an infinite loop.
- Your program will run to completion.

In the first case you can find out where your program aborted and, probably, why. In the second case you can stop your program (by typing `C-c`) and then find out where your program is looping and, probably, why it is looping. In the other case you probably want to set breakpoints.

Executing your program	
`run`	beginning executing the program (can be followed by any command line arguments)
`cont`	continue executing the program
`step`	execute a single line (and step into functions) you can also execute n lines by `step n`
`next`	execute a single line (but do not go into functions) you can also execute n lines by `next n`
`finish`	finish executing the function you are presently in and stop
`until`	continue running until a source line past the current line is reached; this is very useful for getting through `for` or `do` loops
`return`	return *immediately* from the function you are presently in and stop; this cancels the execution of the remainder of the function, as opposed to `finish` which finishes executing the function and then returns; this is a very dangerous command and should be used with great care

Exiting `gdb`	
`quit`	quit the debugger

Odds and ends	
`file`	this file is the executable to be debugged `file` *program_name*
`help`	get help on any command

Print something out and moving around	
`list`	print out the source code in the current function or any other function `list` `list` *first_line_number* , *last_line_number* `list` *function_name*
`print`	print out one variable or array `print` *variable_name*
`whatis`	print out the type of the variable `whatis` *variable_name*
`where`	print a traceback from the current function up to the main program
`up`	go up one function from the current one (useful for printing variables in the function which has called the function you are currently executing)
`down`	go down one function from the current one (usually used after `up`

Breakpoints	
`break`	stop execution (temporarily) at some location (i.e., set a breakpoint) `break` *line number* `break` *function_name*— to stop at the first line of the program do in C or C++ `break main` in Fortran `break MAIN__` or `break main` or ???
`info breakpoints`	shows all breakpoints
`delete` `breakpoints`	delete one or more breakpoints by using the number of the breakpoint(s) (if no number is given, `gdb` will ask if you want all breakpoints deleted)

Change the value of a variable	
`set`	change the value of a variable `set` *variable_name* = *value* or `set variable` *variable_name* = *value*

Searching for string	
`search` (or `fo`)	search forward for a string in the current file `search` *string* or `fo` *string*
`reverse search` (or `rev`)	search backward for a string in the current file `reverse search` *string* or `rev` *string*

Set a watchpoint	
`watch`	set a watchpoint for a variable that is, `gdb` stops whenever the value of a variable changes (which is useful if a variable suddenly takes on an unexpected value)

4.2.3 Ddd, a GUI front end for gdb

Although `ddd` can do many things, it is easiest to think of it as simply a way to use gdb with a mouse. In Figure 4.1 we show a sample example using `ddd`. Note that there are actually **two** distinct windows shown: the full window, which we will call the *ddd window*, and a small window on the right whose first line is "Run". This small window contains a number of "basic" `gdb` commands which can be executed by simply using the mouse, so we will call this the *command window*. It is shown inside the larger window, but it can be moved outside if desired.

`Ddd` was executed by

```
ddd testcode.e
```

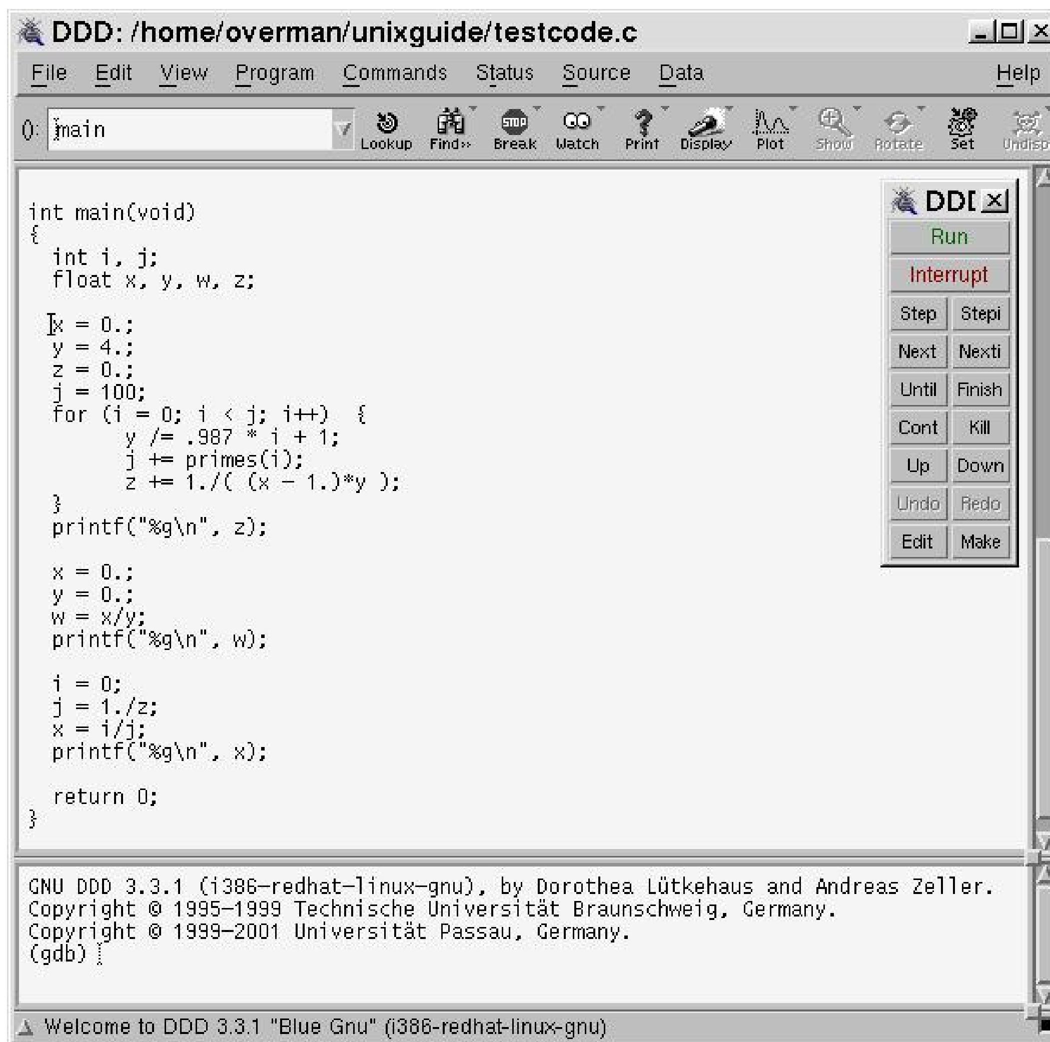


Figure 4.1: A sample `ddd` run.

When it begins executing, it shows the end of the source file containing the main program of the executable file. Notice that the ddd window is divided into two pieces: the top, called the *source window*, shows the program listing and the bottom, called the *debugger console*, is running `gdb`.

You can use `gdb` directly by entering commands in the bottom window. Of course, you can also use the command window. In addition, you can use the menu bar (i.e., the top line which consists of "File ...") and the tool bar (i.e., the second line which contains the icons). Some of the items in the tool bar require arguments, which you enter using the *argument field*, which is at the left end of the tool bar.

For our purposes, you will normally want to set breakpoints and then execute the code. To set a breakpoint, find the line of source code you want to stop at and click the cursor **to the left** of the line of code. Then click the icon "Stop" and a breakpoint will immediately appear as a stop sign. Then run the code using the command window.

4.2.4 Basic dbx commands

Executing your program	
`run`	beginning executing the program (can be followed by any command line arguments)
`cont`	continue executing the program
`step`	execute a single line (and step into functions) you can also execute n lines by `step n`
`next`	execute a single line (but do not go into functions) you can also execute n lines by `next n`
`return`	finish executing the function you are presently in and then stop

Exiting `dbx`	
`quit`	quit the debugger

Odds and ends	
`debug`	this file is the executable to be debugged `debug` *program_name*
`help`	get help on any command

Print something out and moving around	
`list`	print out the source code in the current function or any other function `list` `list` *first_line_number* , *last_line_number* `list` *function_name*
`print`	print out one variable or array `print` *variable_name*
`whatis`	print out the type of the variable `whatis` *variable_name*
`where`	print a traceback from the current function up to the main program
`up`	go up one function from the current one (useful for printing variables in the function which has called the function you are currently executing)
`down`	go down one function from the current one (usually used after `up`

Breakpoints	
`stop`	stop execution (temporarily) at some location (i.e., set a breakpoint) `break` *line number* `break` *function_name*— to stop at the first line of the program do in C or C++ `break main` in Fortran `break MAIN__` or `break main` or ???
`status`	shows all breakpoints
`delete`	delete one or more breakpoints by using the number of the breakpoint(s); `delete all` deletes all breakpoints.

Change the value of a variable	
`assign`	change the value of a variable `set` *variable_name* = *value* or `set variable` *variable_name* = *value*

Searching for string	
`search`	search forward for a string in the current file
`bsearch`	search backward for a string in the current file

Set a watchpoint	
`trace`	set a watchpoint for a variable that is, `dbx` prints out whenever the value of a variable changes (which is useful if a variable suddenly takes on an unexpected value)

4.2.5 Basic ldb commands

Executing your program	
`run`	beginning executing the program
`cont`	continue executing the program
`step`	execute a single line (and step into functions)
`next`	execute a single line (but do not go into functions)
`finish`	finish executing the function you are presently in

Exiting `ldb`	
`quit`	quit the debugger

Odds and ends	
`file`	this file is the executable to be debugged `file` *program_name*
`help`	get help on any command

Print something out and moving around	
`list`	print out the source code in the current function or any other function `list` `list` *first_line_number* , *last_line_number* `list` *function_name*
`print`	print out one variable or array `print` *variable_name*
`whatis`	print out the type of the variable `whatis` *variable_name*
`where`	print a traceback from the current function up to the main program
`up`	go up one function from the current one (useful for printing variables in the function which has called the function you are currently executing)
`down`	go down one function from the current one (usually used after `up`

Breakpoints	
`break`	stop execution (temporarily) at some location (i.e., set a breakpoint) `break` *line number* `break` *function_name*— to stop at the first line of the program do in C or C++ `break main` in Fortran `break MAIN__` or `break main` or ???
`info breakpoints`	shows all breakpoints
`delete breakpoints`	delete one or more breakpoints by using the number of the breakpoint(s) (if no number is given, `ldb` will ask if you want all breakpoints deleted)

Change the value of a variable	
`set`	change the value of a variable `set` *variable_name* = *value* or `set variable` *variable_name* = *value*

Searching for string	
`search`	search forward for a string in the current file
`reverse search`	search backward for a string in the current file

Set a watchpoint	
`watch`	set a watchpoint for a variable that is, `ldb` stops whenever the value of a variable changes (which is useful if a variable suddenly takes on an unexpected value)