THE UNIVERSITY OF MELBOURNE

Department of
Computer Science
&
Software Engineering

Student Manual

Volume A

2007 Edition

Acknowledgements The following Staff have been the main contributors during the evolution of the Student Manual:

Sarah Ardu	Mark Bates	Christopher Boek	Joey Chua
Mike Ciavarella	Brian Coogan	Jamie Curmi	Philip Dart
Gill Dobbie	Paul Dunn	Peter Eden	Wai-keong Foong
Stewart Forster	Kevin Glynn	Rex Harris	Margaret Hassall
David Hornsby	Lorraine Johnston	Roy Johnston	Christopher Jones
Teresa Keddis	Kathleen Keogh	Lars Kulik	Michael Lawley
John Lenarcic	Rod Logan	Olivia March	Chris McCarthy
Alistair Moffat	Ed Morris	Emma Norling	Virginia Norling
Michael Paddon	Heather Payne	Bernie Pope	Ian Richards
Nicole Ronald	Cindy Sexton	John Shepherd	Harald Søndergaard
Liz Sonenberg	James Thom	John Torriero	Jeroen van den Muyzenberg
Terence Wong	Justin Zobel		

This edition of the Student Manual was edited by Lars Kulik.

Thanks are also due to various people at the Universities of Sydney and New South Wales, our academic support programmer Adam Hendrix, our system administrators, Ling Shi and Aaron Burnham, our safety officer Cindy Sexton, and Technical Services Manager Andrew Peel.

If you have any suggestions for improving this manual, or find an error, send an e-mail to:
`sm-editor@csse.unimelb.edu.au`.

For continuing updates and frequently-asked questions on this manual, visit:
`http://www.csse.unimelb.edu.au/teaching/support/studentmanual/`

Contents

A Using this Manual

Volume A of the Student Manual is intended to introduce first year students to the computer systems used in the Department of Computer Science and Software Engineering (CSSE). It will also be of invaluable assistance to students undertaking a graduate diploma within the department. Volume B is intended for use by later year students.

Each student is expected to keep the appropriate volume of the Student Manual on-hand during laboratory sessions. Printed lecture notes and assignments may also reference it. Students are not expected to learn this manual from cover to cover, although they should eventually become familiar with the topics covered here.

This manual refers to the computing facilities maintained by the CSSE department, located in the Northern half of the ICT (Information and Communications Technology) building, 111 Barry St., Carlton. Other facilities provided elsewhere in the University are not described in this manual.

If you are unsure about any details, ask your tutor or demonstrator, or visit the Helpdesk (see Section D for Helpdesk hours and location).

1 On-line Documentation

The authoritative documentation, rules and policies for use of the department's facilities is on the department webpage at `http://www.csse.unimelb.edu.au/`. Students should become familiar with this material. Important sections are noted throughout this manual.

Also note that there are still instances where the department's pre-2006 domain `http://www.cs.mu.oz.au` may be in use.

2 Errata and FAQs

For continuing updates and frequently-asked questions (FAQs) on this manual, visit:

`http://www.csse.unimelb.edu.au/teaching/support/studentmanual`

If you find an error in this manual, or if you have suggestions for improving it, send an e-mail to: `sm-editor@csse.unimelb.edu.au`.

B Principles of Responsible Student Behaviour

Students in the Department of Computer Science and Software Engineering have access to extensive computing facilities for the purpose of completing practical work associated with their course. These facilities include laboratories, communications networks, personal computers, UNIX servers, as well as associated software, files, and data.

These facilities offer many opportunities for the sharing of information. With the ability to share facilities and information comes the responsibility to use the systems legally and to act in accordance with high standards of honesty and personal conduct. Every student in the Department of Computer Science and Software Engineering is expected to respect the rights of others and to use computing facilities appropriately.

Every student will sign a declaration form when applying for their computer account, and you should read this form carefully *before* signing it.

This document describes general principles of responsible student behaviour that have been adopted by the Department of Computer Science and Software Engineering. The University of Melbourne Computing Rules relating to the use of university computing and network facilities are listed in the Student Diary. Students found to be acting in breach of these rules and principles will be penalised.

The computer account declaration, rules for acceptable behaviour in laboratories and ICT building generally, and guidelines for considerate use of shared computing resources are available from `http://www.csse.unimelb.edu.au/local/tech/policies/`. Students should read these documents carefully.

Any questions concerning the interpretation of these principles should be directed to your lecturer or to the Head of the Department of Computer Science and Software Engineering.

Intended Use of Computing Facilities

Students are authorised to make use of computing facilities and resources provided by the University solely for academic purposes directly related to their course of study. As computing facilities and resources are limited, all students should be considerate of others.

- Before using any University computing facilities, e.g., computing equipment, printers, or software, it is the responsibility of each student to find out the conditions of use of the particular facilities they wish to use and to ensure that they have the required authorisation.
- In particular, students may use the Internet (i.e., email, web browsers, etc.), database, text processing, and other programs, provided it is for academic purposes as above and consistent with these principles.
- Computing facilities and resources provided for work in a particular subject should not in general be used for work in other subjects.

Privacy and Security

Students should take all reasonable precautions to protect their own accounts, files, disks, printouts, and other information from unauthorised use, and should also respect the individual privacy of other users.

- Students are responsible for their computer accounts, files, disks, and printouts. They should take all reasonable precautions to prevent unauthorised use of their accounts, files, and disks by other users. Such precautions include care in the choice of passwords and in the setting of file access permissions. System-provided protection features can be used for this purpose.
- Students should not give authorisation to others to use their accounts, files, or disks unless specifically required to do so as part of their course.
- Students who believe their accounts, files, or disks have been compromised, i.e., accessed or able to be accessed by an unauthorised user, should notify the academic support programmer or system administrators immediately.
- Students should not attempt to access the account, files, or disk of another user without explicit authorisation from the other user, and then only if they are working on a joint project.
- Although students may sometimes be negligent or naive in revealing an account name or password to another person, this does not constitute authorisation for that other person to use the account.
- Normally, setting file access permissions to allow public or group access is considered authorisation for others to access those files. However, as many students do not understand these features, other students should not assume they have authorisation to access unprotected files, unless they have explicitly been given authorisation.
- Students should not use or distribute information not intended for use or distribution by its owner. This applies to passwords, files, listings, private notes, written assignments, and so on.
- Students are only permitted in University buildings during authorised opening hours.

Computer System Integrity

Students should not act intentionally to interfere with or to compromise the integrity of a computer system.

- Students should not attempt to restrict or deny access by authorised users to computing facilities. In particular, students should not tie up computer resources by game playing, listening to music or news streams, or other non-academic applications, by sending frivolous or excessive messages, or by using printing facilities inappropriately.
- Students should not use the account of another user, impersonate another user in communication, attempt to capture or crack passwords or encryption, destroy or alter data or programs belonging to other users, or subvert access restrictions such as quotas associated with users' accounts.

Intellectual Property Rights

Students must abide by any restrictions that are associated with the software and/or data that they use.

- Some software and data that reside on file systems that students may access are protected by copyright and other laws, and also by licenses and other contractual agreements. Students must not breach these restrictions.

- Generally, the purchaser of a computer program does not become the owner of the program, but is licensed to use it in accordance with the conditions of sale. Students must abide by any prohibitions that pertain to the use of any software that they are authorised to access. These may include restrictions against copying programs or data for use at another site, against use for non-educational purposes, and against disclosure of information (e.g., program source code).

- Students may not use University facilities for making or running unauthorised copies of software. There will be an automatic fine imposed by the Department for software piracy.

- Students undertake to keep confidential any disclosure to them by the University of software (including methods or concepts utilised therein) licensed to the University for use on its computers and they hereby indemnify and hold harmless the University against any claims of whatsoever nature arising out of any disclosure on their part to another of the said software in breach of this undertaking.

Duty of Care

The University and students have a responsibility to cooperate to provide a safe environment for teaching and learning. The University has a duty of care to students, and students have a duty of care towards each other and the academic and general staff to ensure everyone can learn in a safe environment. Action against students who act inappropriately will be taken in accordance with Statute 13 of the University on Student Discipline.

Specific details regarding health and safety are described in the next section of this manual.

Any Safety questions or concerns about safety should be directed to the Department Safety Officer and/or the Head of Department.

Collaboration and Cheating

Students may discuss their course work with each other, and are encouraged to do so. However, students should only submit work that they have done themselves.

- Work submitted by a student should be the result of that student working substantially independently.

- Students should not use material (e.g., files, listings, hand-written notes) obtained from another student with or without their permission. Material obtained from textbooks or lecture notes may be used, provided appropriate attribution is given.

- Students should not provide another student with material that substantially assists that student in completing work he or she is expected to perform independently.

- Any student who is caught cheating will be penalised in accordance with the University discipline procedures.

Further information relating to the University's position with regard to plagiarism may be found at `http://www.services.unimelb.edu.au/plagiarism/`.

Disciplinary Action

University Penalties

The University is empowered to penalise students for improper behaviour, including withdrawal of access to University computing facilities, failure in an assignment or a subject, imposition of fines, and suspension or expulsion from the University.

The Department has decided that the minimum penalty imposed for software piracy will be a fine of $100.

Crimes (Computer) Act 1988

Under this Victorian Government legislation, Computer Trespass is a criminal offence and possible penalties include a term of imprisonment. Computer Trespass includes gaining access to, or entering, a computer system or part of a computer system without lawful authority to do so. Computer Trespass is an offence regardless of whether or not the trespasser gains, or intends to gain financial benefit, or alters or damages, or intends to alter or damage the files or programs.

Alistair Moffat
Head, Department of Computer Science and Software Engineering
February, 2007

C Health and Safety in the University

The University of Melbourne considers the safety of the staff and students to be of paramount importance. We have in place a continual program of risk assessment and development of procedures to minimise the dangers to the users of University facilities. The program named SafetyMAP (Safety Management Achievement Program) covers a number of areas.

Emergency Procedures Developing procedures to deal immediately with a range of emergencies as they occur.

Personal Security Improving the safety of people around the University buildings and grounds.

Health Risk Minimisation Minimisation - Making the facilities safer for all users to avoid longer-term chronic health problems.

Together with SafetyMAP the University also incorporates an EMS (Environmental Management System) which introduces ways to improve and create a sustainable environment for all.

All students have a responsibility for adopting safe work and study practices, and are required to comply with all University and Departmental rules and procedures (see University EHS manual at `http://www.unimelb.edu.au/ehsm/` and Department safety pages `http://www.csse.unimelb.edu.au/local/admin/safety/`) which relate to environment, health and safety.

All Students . . .

- must report all hazard and injuries to your Supervisor, Department or Faculty General Office or Sports Centre;
- must not wilfully place at risk the health or safety of any other person at the University;
- must not wilfully or recklessly interfere with or misuse anything provided in the interests of environment, health and safety or welfare at the University.

The use of certain facilities may require that students provide some items of personal protective equipment- see Procedure 8.3.4 Selection and issue of Personal Protection Equipment (PPE) in the University's EHS Manual.

1 The ICT Building

In this document we will be making some specific reference to the procedures in place in the ICT building. Note that every University building has its own set of procedures and you should endeavour to familiarise yourself with them when using other facilities. The safety manual for the Department of Computer Science and Software Engineering can be found at `http://www.csse.unimelb.edu.au/local/admin/safety/resources/ICT_EHSS_Manual.pdf`. A hard copy is posted on the EHS notice board (nearest the amenities block) on Level 4.

1.1 Emergency Evacuation

The emergency evacuation signal is a loud repeated beep. Evacuate the building as soon as you hear it on your floor. Everyone must leave the building (via the nearest safe exit) in an orderly manner and proceed to the assembly area in the park across Barry St.

- Move as far away from the building as you can to avoid any shattering glass and falling debris. Additionally, this allows emergency personnel easier access to the building.
- **Do not use the lifts** unless physically incapable of using the fire escape stairs. Exit by the nearest fire escape or safe exit or as otherwise directed by emergency personnel. Summon the assistance of a Fire Warden if you need help.

During office-hours, Fire Wardens will normally direct and supervise an evacuation. After hours — take the initiative to exit yourself or follow the directions of any lecturers or lab supervisors present. It is wise to know the evacuation routes before they're actually needed. Floor maps indicating exit routes are displayed on notice boards nearest the amenities block on all floors.

1.2 First Aid

First aid is the initial care given to an injured person. It cannot take the place of skilled medical attention. First Aid kits (with basic supplies and no medications) are located with the Building Supervisor (Grant Young) on ground floor, the CSSE Help Desk on level 1 (during semester only), and in the amenities foyer (under the sink) on levels 4 and 6.

During working hours, the Department of Computer Science and Software Engineering has staff able to provide basic first aid. Contact the main office on Level 4 or one of the First Aid Officers. First Aid Officers names are posted on notice boards nearest all amenities blocks and on the Department "Safety" web-site. In addition, University Security personnel have Level 2 Workplace First Aid qualifications and can be contacted 24 hours a day, 7 days per week on extension 46666 or Free call 1800 246 066.

The closest medical centre is the Student Health Service located at 138 Cardigan Street, Carlton. The Royal Melbourne Hospital has a 24 hour casualty department (on Grattan St, just across Royal Parade), and the nearest after hours (between 8.00am - 10.00pm) private medical clinic is Betta Health Medical Centre, 30 Sydney Road, Brunswick (ph. 9380 2866).

An Ambulance should be called (phone 000) for casualties involved in serious incidents that are life-threatening or require urgent hospitalisation (including serious eye injuries). Ensure accurate location details are given to emergency services and where possible - have someone meet them at Street level to usher them urgently to any casualties.

2 University Security - First Point of Contact

If there is some emergency situation anywhere on University premises, whether it be a building emergency, security incident or where there is personal danger - University Security can be contacted 24 hours a day on extension 46666 or Free call 1800 246 066. They can also be contacted via the many Security intercoms located in ICT as well as blue pole intercoms located around the main campus. Security will respond and contact the appropriate authorities in an emergency.

3 Occupational Overuse Syndrome (RSI)

Occupation Overuse Syndrome (OOS), which was previously known as Repetitive Strain Injury (RSI), is still occurring in the University community. It is a painful and debilitating condition which can take a long time to heal, so prevention is still the best cure! Frequent breaks, good posture and appropriate

working conditions help avoid developing OOS. Because OOS is a complex condition and people react differently to work-place stresses, if you have any serious concerns for your own health, it is strongly recommended that you seek medical advice.

D Support Services

There are a variety of support services for students in the ICT building offering a variety of different levels of support.

At the ICT building, the following sources of help are available:

- Your demonstrator, during your scheduled laboratory class.
- The First Year Centre, room 1.19 on level 1. This centre is open throughout semester for around 10 hours a week (for exact hours, check the notice on the door, or ask your lecturer, tutor or demonstrator). The staff of this centre are current tutors of first-year subjects, so are well equipped to deal with any queries about first-year subjects. In addition, since they are extremely competent senior students in the department, they may also be able to assist with queries from later year students. For more information, visit the First Year Center web page at:
  `http://www.csse.unimelb.edu.au/teaching/support/consultcentre/fyc/`
- The student Help desk is open during Semester between 9am and 6:30pm in room 1.30 on level 1 for help with problems on non-subject-specific material, such as accounts, printing problems and machine difficulties.
  `http://www.csse.unimelb.edu.au/teaching/support/helpdesk/`

Finally, you can try the following sources:

- The head tutor for each subject is available for discussion of more specialised programming problems and for other course-related material.
- The lecturer for each subject can be consulted about course-related material and other problems.

Both the head tutor and lecturer will usually have timetables on the subject website and on their office doors indicating the times when they are available.

In addition, for support in matters which are not subject specific, the University provides a number of services, such as the learning skills unit, student counselling, etc. Refer to your student diary to find out exactly which services are available, and how to access them.

1 Utilising the Support Services

Relevant subject notes and text need to be read before consulting with a tutor, and any supplied code needs to be worked through. There is also an expectation that, except in special circumstances, students will have attended all lectures and tutorials.

Students seeking assistance on programming projects should bring a *well-formatted* and *current* printout with them, and be able to describe their problem as well as their own efforts to solve it.

Increasing independence is expected of students as they progress through their course. By the end of first year, students are expected to perform routine debugging of programs on their own, and are thereafter expected to show increasing initiative in utilising on-line and printed resources. Students at all levels are encouraged to try out and assess their own ideas.

While the support systems are available to help students, excessive reliance on them for project work is discouraged in later years.

2 Systems Support

The computer systems are maintained by the Technical Services section of the department. Their homepage (`https://www.csse.unimelb.edu.au/local/tech/`) has links to:

- Policies which you should understand, including guidelines for acceptable behaviour in the laboratories, use of the Internet and the systems, and the account declaration.
- Systems documentation describing the system configuration.
- Contact information for various types of requests.

2.1 Reporting faults

Students are encouraged to report faulty equipment. You can do this by:

- running the `fault` command at a UNIX shell prompt,
- filling in the form at `https://mailhost.csse.unimelb.edu.au/rts/SelfService/Create.html?Queue=fault`[1], or
- sending email to `rts+fault@csse.unimelb.edu.au`. Remember to specify the room number, identification number written on the equipment (e.g., PC676), and the nature of the fault.

[1] There is a link to this URL from the Technical Services *requesting support* page at `https://www.csse.unimelb.edu.au/local/tech/contact/`

E Schedule for Acquiring UNIX Skills

This section details at what stage in your course we expect you to learn certain UNIX commands. There are other commands not listed here which you will probably also find useful, but this in the bare minimum that you will have to learn. Most of the commands that you will need to learn in first year are covered in the next two sections of the manual.

Type of Command	**During 171/151**	**During 172/152**	**During 253**	**During 252/254**
Access, control, shells:				
Login, logout	login, logout `^D`, exit, passwd, rlogin			.bash_profile, .bashrc
Job Control	`^C, ^Z`, fg jobs	ps, kill	nice	&, bg
Environment, Shells	term, stty	functions		bash, tcsh, .bash_profile, .bashrc, echo, export, set
Files and directories:				
Editing	vi	vi		
Files, Filters	cp, mv, rm	chmod, diff, wc, ispell	ln, grep,	cat, sort, tr, rcs, cvs
Displaying text files	less, more, `^S, ^Q`		head, tail	
Printing	lpr, lpq, lprm			
Directories	cd, ls, mv, pwd, mkdir, rmdir	`. , ..`	ln -s	
Programming environment, compilers, debuggers:				
Compilers, debuggers interpreters	gcc	gnuc[1] gdb or dbx, error hugs	objectcentre, lint	ldd, make

Type of Command	During 171/151	During 172/152	During 253	During 252/254
Command history, re-direction and piping:				
Entering a Command		`!, !!, ^^,` history		
Re-direction, Piping		`>,>>,<,` `2>&1, |`		
Information:				
Manual	man (intro.)	man, apropos, whatis		
Information	quota, date		who, where, which	find, locate, whereis, help
Mail	pine, elm or mutt			
News	nn			
Internet, www	lynx	ftp	scp, ssh, mozilla	
Other:				
Submitting projects	submit[1], verify[1]			
Windows				Windows management

Commands, to be learnt when needed (but definitely by the end of 3rd year):

bc, cal, finger, chfn, emacs, cut, paste, du, fmt, pushd, popd, tar, umask, gzip

Notes:

1. Special Melb. Univ. commands and scripts: gnuc, submit, verify

F Introduction to UNIX

UNIX is a multiuser operating system that is in popular use at academic institutions and in industry all around the world. It provides an excellent environment for program development. Hence it is a natural choice for use in the Department of Computer Science and Software Engineering.

Despite being developed in 1969 by Ken Thompson at AT&T Bell Laboratories, UNIX still enjoys widespread use — a testament to its purposeful design and functionality. Today, there exist several 'flavours' of UNIX which run on a variety of different types of computers. The computers you will be using at CSSE are running Solaris. Other UNIX flavours commonly used by staff and research students in CSSE are Linux and BSD (Berkeley Software Distribution).

Although UNIX has the reputation of being unforgiving to the beginner, perseverance is rewarded with the efficiency of completing tasks that the user gains with experience.

These notes give a practical hands-on introduction to the UNIX operating system, and to some of the more important UNIX commands.

It is expected that you will go through these examples whilst you are using one of the University's computers — in hands-on, interactive learning. So before you can start you will need to have an account on the computer system that you are to use. This means that you will have to complete some necessary administrative matters, then get your *username* (or login name) and your first *password*, to enable you to gain access to the computer. Most of the laboratories nominated for this work contain computers running Microsoft Windows, from which you can access the UNIX servers using a *terminal client* such as `sshclient`.

Once these preliminaries are completed, you will be ready to work through the examples provided and to run the various commands. Study the description of each command, the output from each command, and try to understand what each command did before you progress to the next command. Make notes and keep these for your future use.

Many books about UNIX are available for reference or for purchase. If you are buying a text, it is suggested that you might obtain one that not only provides a good introduction to the UNIX system, but which is also sufficiently comprehensive to serve as a reference text well into the future.

Some texts that have been found to be useful are:

- *A Student's Guide to UNIX*, Harley Hahn (McGraw-Hill, 1993).
- *A Practical Guide to the UNIX System, 3rd Edition*, Mark G. Sobell (The Benjamin/ Cummings Publishing Group, Inc., 1995).

When you *log in* to the UNIX system, you will be using a *shell*. This shell is simply a program which lets you run commands. On the UNIX system, there are many different shells available, each of which works in slightly different ways, and the shell the we expect you to be using is *bash*. Some of the commands in some of the sections of this manual will *not* work in the expected manner if you are using a different shell.

The following books are about certain aspects of UNIX — the first is a reference for the UNIX shell that you will be using, and the second is a reference for the text editor you will use.

- *Learning the BASH Shell*, Cameron Newham & Bill Rosenblatt (O'Reilly & Associates, Inc., 1995).

- *Learning the vi Editor*, Linda Lamb (O'Reilly & Associates Inc., 1990).

Remember, if you run into trouble at any time, first check the section on *UNIX Troubleshooting* in this manual, and if that does not give the solution to your problem, ask your classmates, your demonstrator, the Academic Support Programmer or another staff member.

1 Starting with UNIX

1.1 Before you start

Before you log into the computer, make yourself familiar with your terminal or workstation and its keyboard. What type of terminal or workstation is it? Are you required to leave the computer switched *on* when you finish your session, or are you requested to switch it off; and if the latter, where is the on-off switch? What is the name of the computer (or host) that you will be using?

Before you go through these examples, make sure you read them through and understand how to log out of your session. (Always, before you start a new program or activity, you should know the proper way to finish, or quit, or exit from that activity.)

Note that:

- Every command is completed by pressing the RETURN or ENTER key.
- If you make a typing mistake as you type in a command, you can correct it (before pressing the RETURN or ENTER key), by using either the BACK SPACE or the DELETE key, and then re-typing the command. (If you cannot find a key that works, refer to the section on *UNIX Troubleshooting*.)
- UNIX is case-sensitive; the file names `mary`, `Mary` and `MARY`, if used, will represent different files. There is a UNIX command called `cat`, but not one called `Cat`.
- When you want to finish your session at the computer, make sure you follow the instructions given below describing how to log out.

1.2 Logging in

Before you can log in and start to try out these examples, you must already have obtained both your login name and your (initial) password.

Logging in via a terminal client

You connect to the UNIX server using a *terminal client* program. Examples are `sshclient` or `PuTTY` on Windows, `xterm` on Xwindows, or `ssh` from another UNIX server or Cygwin on Windows.

You will need to specify the address of the UNIX server you are connecting to. `student-random.csse.unimelb.edu.au` will connect you to one of the UNIX servers at CSSE for student use. Once the connection is established, the UNIX server will ask for your login name (or username) and your password. (You complete each entry by pressing the RETURN or ENTER key.)

Your login name is the way you are known on the computer; each user has a unique login name. Your password will *not* appear on the screen when you type it in. It provides a measure of security against others accessing and using your computer account. You are the only person who should know your password.

Logging in from an X terminal

Several laboratories at CSSE contain X-terminals. These provide a login to the UNIX servers which allows you to run graphics applications in addition to the text-based commands which may be run via a terminal client.

The X-terminal screen should display a box in which you type your login name. It will then ask you to type in your password. After logging in successfully, you need to start the `xterm` program to open a terminal window in which to type UNIX commands.

Logging in from a Windows PC

Other laboratories at CSSE contain personal computers running Microsoft Windows. The login screen asks for your login name, password and domain, in which you should type `CSSE`. When you have successfully logged into Windows, your files on the UNIX server will be available as drive `H:`.

You can then log in to the UNIX servers by running the `sshclient` terminal client. Or you can run the `Xwin32` X-windows emulator.

Each time you log in, the "message of the day" will be displayed to you. This contains information about the computer system. It may tell you about downtime (i.e. times that the computers will not be available), and other important things. You should make sure you read it when you log in.

After logging in successfully, and after executing each of your commands, UNIX will display a command prompt of some form. This prompt signifies to you that the machine is awaiting input from you, through the keyboard. If this prompt does not appear after executing a command, type the interrupt character CONTROL-C (often written as `^C`) by holding down the CONTROL key and pressing the C key.

In these notes, we will not mention this prompt again.

Try entering these commands:

```
date
cal 3 1998
who
ls
w
man cp
clear
set
pwd
```

(To *enter the command*, type the word(s) (e.g. `date`) and press the RETURN or ENTER key.)

These are just a few of the many commands available to you in UNIX. We will look at these commands and many more in later sections.

1.3 Changing your password

You need to know how to change your password.

The instructions given to you when you obtain your login name and your initial password will instruct you to change your initial password to one of your own choosing when you log in for the first time. You should change your password fairly regularly (every one or two months is a good idea), to prevent other people from guessing it. Your account will be disabled if you do not change it often enough (but you will be given advanced warning).

Restrictions are placed on the *word* that forms your password — this will vary depending upon the system you use, but generally you should try to use a combination of lower and upper-case letters, numbers and non-alphanumeric characters. If the computer considers your password too easy to guess, it will ask you to choose a different one.

Here are some hints on passwords:

- Do not choose words that appear in the dictionary
- Do not use given or family names
- Do not use advertising slogans or 'catch' phrases
- Do not use car registrations or birthdates
- Do not tell anyone your password
- Do not write your password down

There are restrictions placed on what you can use for your password to make it more difficult for your password to be guessed or worked out.

As mentioned above, your password must be known only to you, and not to any other colleague, staff member or other person.

To change your password, enter the command:

```
chpw
```

and then respond to the prompts as they appear on your screen. To double-check, the system requires you to enter your new password twice; if the two entries are not identical, the computer retains your original password. NOTE THAT for security reasons, neither the old nor new passwords are displayed (*echoed*) on the screen, so they must be typed carefully.

Be sure to remember your new password, without writing it down, because it is the new password that you must use the next time you log in. Note that it may take up to 10 minutes before Windows starts to use your new password.

If you forget your password and absolutely cannot remember it, you should visit the Helpdesk for assistance.

1.4 Logging out (logout)

When you wish to end your session with the computer, you must log out from the computer. To logout from a UNIX text-based terminal session, enter one or other of the commands:

```
logout
```

or

```
exit
```

If you are using an X Terminal, you will need to find `logout`, `exit` or `quit` on one of your pulldown menus. (To use your pulldown menus, move the mouse pointer *outside* any windows on your screen, then hold down each mouse button in turn. You will probably have a different menu on each mouse button.)

If you are using a windowsXP PC, select `shutdown` and click `logout`.

Try to log out now. You should make sure that you have logged out successfully. (Generally, if you have logged out successfully, the terminal will be displaying the same thing that it did when you first arrived.) If you are unsure whether you have logged out correctly, check with your demonstrator.

If at any time you cannot log out, refer to the section on *UNIX Troubleshooting* in the back of this manual, or get help from your demonstrator, or a system administrator. **Never** leave your terminal without logging out. If you do this, other people will be able to use your computer account, and may cause a lot of heartache to you (by deleting your files, or other more serious things). Remember that you can be held responsible for whatever is done from your account, even if someone else was using it.

1.5 Summary

Here are the main points of this section again in brief:

- To sign onto the system, you must supply your *username* and *password*
- To sign off the system, use either **logout** or **exit**.
- To change your password, use the command: **chpw**.

Now log in again, so that you can continue, but **remember to log out before you leave the computer today!**

2 Getting Help in UNIX

2.1 The 'man' command

Enter the command:

```
man man
```

What did this do? To move through the displayed text, press the **space bar** on your keyboard. To exit, press **q**. Try entering the command:

```
man mv
```

Can you guess what will happen when you use the `mv` command? We'll be looking at its usage in a later section.

The `man` command can be used to get information on a command that you already know the name of. Try the command:

```
man
```

You should have a few lines of text starting with "Usage: ..." on your screen. This is the computer's way of telling you that you have used the command incorrectly, and a hint about how it should be used. If you use a UNIX command incorrectly you will usually get a message like this. Usually it is because you have given too many or too few *arguments* to the command. In this case, the command expects an argument which is the name of the command you want to find out about.

Enter the command

```
mam
```

This is an example of what happens if you accidently mis-type a command. (Occasionally if you mis-type a command, you may accidently type the name of a different command that *does* exist. — Be aware of this, since the command you do actually run can have a vastly different purpose from the one you wanted!)

Well, this means that `man` is great if you know the name of the command you want to use. But what about if you do not know the name of a command? In this case, you need to use the `-k` *flag* with the `man`. For example, lets say you wanted to copy a file, but you do not know the command to do it (e.g. if you are used to using DOS, you might expect the command to be `copy file1 file2`). Enter the command:

```
man -k copy
```

If some of the output scrolled off your screen, use the command

```
man -k copy | less
```

instead. The output will pause after displaying each screen, and display the next screen after you press the spacebar. (We'll look at what the | operator does in more depth in the section *Piping and Redirection* and at what the `less` command does in the next section, entitled *Looking at your files*).

You should see a list of commands, all of which have something to do with copying. You then need to find the one that has something to do with copying files. See if you can work out which is the appropriate command. Did you choose `cp`? This is the UNIX command for copying files.

You should know that this method of finding the command you need is not foolproof. For example, say you entered the command `man -k delete`, hoping to find a way of deleting your files. You will not be able to find a suitable command, because this command is stored with a keyword of **remove**

instead of delete. Most of the time, you should be able to find what you need though. If you cannot find the command you need, ask your classmates, a demonstrator, or another staff member.

3 Using Files in UNIX

3.1 Looking at your files (ls, cat, more, less)

You will need to store your programs or scripts as files. If you are using Hugs, the system can create these files for you, but there will be times when you need to access these files from UNIX (for example when you submit your projects). Firstly, we need to create some files. Normally you would do this with an editor (e.g. `vi`), but since you probably have not learnt how to use it yet, we will quickly create a few — do not worry about how it is done below, we will discuss how this is done in the section entitled *Piping and Redirection*.

Enter the commands:

```
man mv > mv.man
```

and:

```
cal 3 03 > Mar2003
```

Now enter the command:

```
ls
```

In case you have not guessed, the command `ls` is used to obtain a listing of the files and/or directories contained in the current directory.

You should see two files listed: `mv.man` and `Mar2003`. (There may also be some other names: these would be files which already existed in your directory area.)

Now let's have a look at the contents of these files. Enter the command:

```
cat mv.man
```

This will display the contents of the file on the screen. Most likely though, most of it scrolled off the top of your screen. To view it one screen at a time, enter the command:

```
more mv.man
```

This will display the file on your screen, one screenful at a time. As before, press the spacebar to see the next page. You can also type: 'q' (which stands for 'quit') to go back to your UNIX prompt before getting to the end of the file; RETURN or ENTER to see the next line of the file; or 'h' to get help. `less` is a command which works in a very similar fashion to `more`. Enter the command:

```
less mv.man
```

and decide which you prefer. From now on, whenever you are instructed to use `more`, you can use `less` if you prefer.

3.2 Moving, copying and deleting files (mv, cp, rm)

Now enter the command:

```
more Mar2003
```

You'll probably notice that there's something wrong with calendar — it's not actually the calendar for this month, it's the calendar for March in the year 03 A.D. Well, lets fix that. Enter the commands:

```
mv Mar2003 Mar0003
```

(those are zeroes, not capital Os) and

```
cal 3 2003 > Mar2003
```

Now enter `ls` again. You should now have these files: `mv.man Mar0003 Mar2003` (and maybe some others). Check the contents of `Mar0003` and `Mar2003` to satisfy yourself that they are correct.

How did we fix it? Well, first we used the `mv` command, to *move* (or *rename*) the file `Mar2003` to `Mar0003`, then we used the `cal` command to get the right calendar for March 2003. (Check the man page for more information on the `cal` command. The calendar for September 1752 is interesting — have a look at it.)

Okay, let's say you've finished using the file Mar0003, so you want to delete it (you should always delete files if you are not going to use them again, but see the warning below). Enter the command:

```
rm -i Mar0003
```

The computer will give you a prompt looking something like this:

```
rm:  remove Mar0003 (y/n)?
```

to which you should respond `y`, as in "Yes, remove it." The `-i` is another *flag*, and a very useful one. The `rm` command, when used without any flags (e.g. `rm Mar0003`) will just delete the file. It will then be gone, and you will not be able to get it back. If you use `rm -i`, at least you have a chance to change your mind!

That takes care of deleting a single file, but what if you want to delete more than one file? You could run `rm` with a list of the files you want deleted, but this could be difficult if you want to delete many files simultaneously. The command (**do not run this right now!**):

```
rm -i *
```

will delete all files in the current working directory, asking for confirmation for each one (because of the '-i' option of `rm`). '`*`' (star) is called a *wild-card*, which runs `rm` with the names of all of the files in the current directory.

Be very careful with what you delete, because under UNIX it is not easy nor even always possible to get it back. Your systems administrators will help you retrieve project material and other course-related work if possible, but this is inconvenient both to yourself and them, so the safest way is to work under the assumption that if you delete something you will never get it back.

The commands `cp` (for copying files) and `mv` (for moving or renaming file) can also use the `-i` flag. When used with these commands, the flag will prevent you from overwriting an already existing file. For example, enter these commands:

```
cal 2 2003 > test
cp -i test Mar2003
mv -i test Feb2003
ls
```

Firstly we created the file. Then we tried to copy the file called `test` to the file called `Mar2003`. Because `Mar2003` already existed, you got a prompt similar to this:

```
cp:  overwrite Mar03 (y/n)?
```

If you responded `y`, the contents of `Mar2003` were replaced with the contents of `test`; if you responded `n`, the contents of `Mar2003` remained unchanged. Next, you moved the file `test` to the file `Feb2003`. This time you were not given another prompt, because there was not already a file called `Feb2003`.

Finally, when you did your listing, you should have had these files: `mv.man`, `Mar2003` and `Feb2003` (and perhaps some others). You should *not* have seen the file `Mar0003` in your listing, because you should have deleted it earlier.

3.3 Printing your files (lpr, lpq, lprm)

There may be times when you wish to have a printout of your work to take home with you, particularly when you are working on your projects. You should be careful not to print out too often, because

- This wastes trees
- You will probably have a printer quota - that is, a limit on the number of sheets of paper you can print out. Once you have exceeded this limit, you will not be able to do any more printing for the rest of the semester. Check with your demonstrator whether you will have a quota this year, and if you do, ask what the command is to check your quota and write it down here:

Printing from your UNIX account means that you will be printing in a networked (i.e. shared) environment. Your print job will be put into a queue of jobs for that printer. This means that the time your document takes to print depends not only on the size of your document, but also on how many people are ahead of you in the queue, so you will probably find that whilst a document may print straight away if you printed it early in semester, the same document may take several times longer if you are trying to print it around project submission time. By this token, please show respect for other users and do not print out excessively large files (remember that quota!) when the demand for the printer is high.

To print out a file or files, you use the `lpr` command like so:

`lpr -P` *printer_name file(s)*

To help you save on paper (and quota!) you can use the `a2ps` command

`a2ps -P` *printer_name file(s)*

Each lab should have its own printer. The printer-name for the printer you wish to use, is given by the room number for the lab that the printer is physically located in. So if you were printing the file `my_proj.c` using the printer in lab UG.09, the command would be:

`lpr -P` *ug09 my_proj.c*

(If you're using `a2ps`, then your command will be: `a2ps -P` *ug09 my_proj.c*

To view the jobs queued at a specified printer, type the command:

`lpq -P` *printer_name*

This will display a listing of the jobs in the print queue and their status for that printer.

It will look something like this:

```
Rank   Owner      Job  Files         Total Size
active bgates     42   memo.out      226157 bytes
1st    slf        43   report.txt    13090 bytes
2nd    foo        44   my_proj.c     18235 bytes
```

If you are the user named *foo*, then you will see that you have a file called *my_proj.c* named as job number *44* which is third in the queue. Also note the *total size* column: the first job in the queue is an order of 10 times larger than the other two, and so it will take longer to print.

If you decide that you do not want a printout after all, you can cancel your job by using the command:

`lprm -P` *printer_name job_number*

Where the *job_number* is found by using `lpq` above.

Using our example, to stop printing `my_proj.c` you'd type:

```
lprm -P ug09 44
```

However, if your file has already started printing, i.e. it's rank is "active", at least *some* of the document (and perhaps the whole document) will still print out, so make sure that you collect it from the printer.

If you'd like to make sure that you can print a file successfully, enter the following commands (but feel free to skip this if you're confident that you will be able to print later when you need to):

```
cal 2003 > 2003
```

and

```
lpr -P printer_name 2003
```

This will give you a printout of the calendar for this year. Try out `lpq` to see your job in the queue. **Make sure you collect your printout!**

This will place your print job into a queue of jobs waiting to be printed out on that printer.

A useful tip is to enter `lpq` before `lpr` to see how many jobs are currently in the queue. If there are several jobs in the queue and you need to leave in a hurry, there is probably no point printing at that time.

Always remember to pick up your printouts as soon as possible: if you leave them lying around or forget to collect them, other students may collect them and copy your work. If you go to collect you work and find that someone has taken it, please notify the tutor-in-charge, in case someone copies your work. (Please make sure, using `lpq`, that your document has actually finished printing before assuming that someone has taken it.)

3.4 Summary

To summarise what we have learnt in this section:

- To obtain a listing of the files in your directory, use the command: **ls**.
- To view the contents of your files, you can use one of three commands: **cat**, **more** and **less**.
- To delete files, use the command **rm -i**.
- To move files from one location to another or rename files, use the command **mv -i**.
- To copy files from one location to another, use the command **cp -i**.
- To print files, use the command **lpr**.
- To check how your file is progressing in the print queue, use the command **lpq**.

- To remove your file from the queue to be printed, use the command **lprm**.

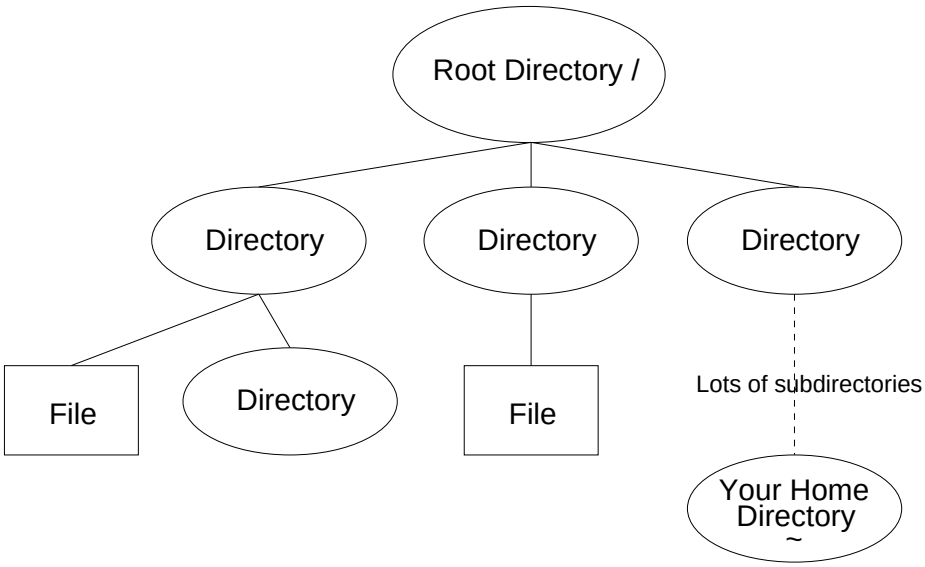
4 Keeping Track of Your Files

4.1 The UNIX File System

So far, we have only encountered text files. In this section we describe the most important aspects of the UNIX file system - that is, how files are organised.

The UNIX file system uses the notion of *files* and *directories* in a tree structure.

Consider the diagram below:



A directory is a special kind of file which can contain other files and directories. In this way, you can organise collections of related files and directories into a single group. This makes it much easier to find relevant files.

A directory which contains some files and/or directories is said to be the *parent directory* of those files and/or directories.

Once you have an account created for you, you will have been allocated a directory. This is your personal work space and is called your *home directory*, also known as ~.

The parent directory of the whole system is called the *root directory*, also referred to as /. Every other file and directory on the system, including your home directory, is contained in a *subdirectory* of the root. The parent of the root directory is itself.

Files are used to store information, whether that be readable text or executable programs.

4.2 Creating and removing directories (mkdir, rmdir)

When you log in to your account, you will automatically be in your home directory. You use it to store your personal files. There are also other directories which are used by the system, which store things such as the programs for the UNIX commands you use. When you first get your account, your home directory will have no subdirectories. All the files you create will be placed in your home directory. You can imagine, if you kept doing this, you would have an awful lot of files in your home directory by the time you finished 3 years (or even 1 semester) of Computer Science. For this reason, you

should create a directory structure *below* your home directory, which will make it easy to find your files when you need them.

Let us try now to create a directory. Enter the command:

```
ls
```

and you should see a listing of the items in your directory. Now enter

```
mkdir intro
```

The command `mkdir` is used to "make a directory". Now again enter

```
ls
```

You should now see *intro* in addition to what was displayed the first time you ran `ls`.

Now try removing the directory *intro* by entering the command

```
rmdir intro
```

Type `ls` again and you should not see the directory named *intro*. `rmdir` will only remove a directory if it is empty; that is, if it contains no files or directories.

In a previous section, you created three files: `mv.man`, `Mar2007` and `Feb2007`. Let's re-create the directory called *intro* and move those files in there. As before, enter the command:

```
mkdir intro
```

Now enter the command

```
ls
```

Your listing of files should look something like this:

```
Feb2007 mv.man intro Mar2007
```

(You may have more files listed than this.) Note that the directory you just created looks exactly the same as the ordinary files in your listing.

Enter the command

```
ls -F
```

Your listing of files should now be something like this:

```
Feb2007 mv.man intro/ Mar2007
```

The `/` after the name `intro` indicates that this is a directory rather than a normal file. The other files are plain text files, so they have nothing after them. Other characters that you might encounter after the file names are `*` (indicating a *binary* or *executable* file) and `@` (indicating a link to another file). This `-F` *flag* for `ls` can be very useful when you are trying to find your way around the file system.

Now we need to move the three files `mv.man`, `Mar2007` and `Feb2007` into this directory. To do this, enter the command

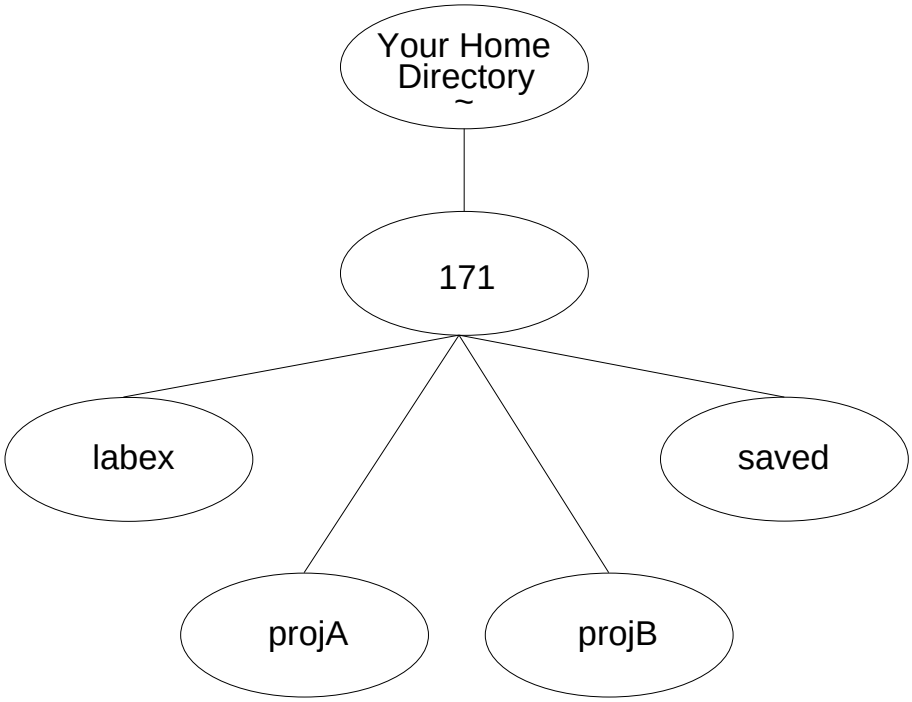
```
mv mv.man Mar2007 Feb2007 intro
```

Do a file listing again. `mv.man`, `Mar2007` and `Feb2007` should not appear in your listing any more. Remember in a previous section we used the `mv` command to rename a file? Here we have used the same command with a slightly different effect. If the last argument *is* a directory name, you can move as many files as you like into that directory, as we have just done. If the last argument is *not* a directory name, there must be only two arguments: the name of the file to be moved, and the name of the file to move it to.

It is a good idea to create a subdirectory for each Computer Science subject you are taking in which you can store the work for that subject. A good name for each directory would be the unit number of the subject (e.g. `171`). If you have already created some files for that subject, move them into that directory. See the section below for more details.

4.3 A suggested directory structure

In order to keep your home directory neat, we suggest a directory structure something like this (the example given is for the subject *433-171 Introduction to Programming*, but you can of course use other names for other subjects):



That is, create a subdirectory for each computing subject you are doing, and create further subdirectories for each project you do in that subject, and one in which you will keep the work you do in your weekly laboratory classes. It is also a good idea to have a subdirectory for each subject called "saved" — see the section entitled *Keeping a backup copy*.

4.4 Moving around (cd, ., .., ~, pwd)

Enter the command:

```
ls intro
```

You should see the files `mv.man`, `Mar2007` and `Feb2007`. What you have just done is obtained a listing of the directory called `intro`. Now enter the commands:

```
cd intro
```

and then:

```
ls
```

You should see exactly the same listing as when you entered `ls intro` before. This is because you have changed directories to the `intro` directory. It is now your *current working directory*. Enter the command:

```
cd ..
```

Now do another file listing. Your listing should now include the `intro` and subject directories (if you've created them), and perhaps some other files. This is because `..` (dot-dot) is an abbreviation for "the directory above the current working directory". There are two other useful abbreviations: `.` (dot), meaning "the current working directory"; and ~ (tilde), meaning "my home directory". Enter the command:

```
pwd                    (print working directory)
```

to display the full name of the directory in which you are currently working. This is the full path name of your home directory, which you can also refer to as ~. Now enter:

```
cd /
```

Followed by:

```
pwd
```

Notice that your current working directory is `/` (slash). This is *not* an abbreviation. As mentioned before, / is the root directory of the system, and all other directories are below it. Now you need to go back to your home directory. How do you do this? Well, you could enter `cd` *`the full path name of your home directory`* (which you saw when you entered the command `pwd` above), but it's difficult to remember the full path name of your home directory. You could enter `cd ~` (remember ~ is your home directory), but it's even easier than that: just enter the `cd` command with no arguments. This is very important to remember: if you ever get "lost" in the system, use `cd` to get home.

4.5 Absolute and Relative pathnames

There are two ways in which you can refer to any file or directory on a UNIX system. One is by specifying the full pathname to that file or directory from the root — it will always start with the root directory /. For example (do not enter this):

```
/home/stude/data/1701
```

This is called the *absolute pathname*.

The other way is to specify the file or directory relative to the one you are currently in. For example look back at the diagram of the suggested directory structure — if you were in the directory *projB* and wanted to move into the directory *projA*, you could enter:

```
cd ../projA
```

Similarly, if you were in the directory *171* and wanted to go into the directory *projB*, you would enter:

```
cd projB
```

This is called the *relative pathname*.

4.6 The Student Data Area

For most subjects, there will be a student data area. This will be referred to in your notes as *student data area*, but you should find out its actual location from your tutor or demonstrator.

Ask your demonstrator where the student data areas are for your subject(s), and write them down here for future reference:

The student data area will be used for a number of purposes. Some of your lab and tute exercises may be partially written for you — in these cases you will be able to copy the incomplete programs from the student data area, rather than typing it all in yourself. You may also find information about your projects in this area. Your tutor, lecturer or demonstrator will let you know about these things as they become available.

Make sure you know how to access files in your student data area. Change your current working directory to the student data area now. Sketch the complete directory tree under this directory, distinguishing between directories and non-directory files. (You can use the `-F` flag with `ls` to do this, as discussed before.)

You will be accessing this directory area regularly during the course of your subject, so it is worth while knowing exactly how it is set up and what will be put into it for your use.

Let's try copying something from the student data area into your current directory. Find out what is in the student data area by doing entering:

```
ls student_data_area_path
```

If there is a file called *copytest* in it, then enter the command:

```
cp student_data_area_path/copytest .
```

Do an `ls` to convince yourself that you have successfully copied the file.

An alternative way of copying the file is to first change directories to the student data area, using the following command:

```
cd student_data_area_path
```

Then copy the file to the `intro` subdirectory of your home directory using this command:

```
cp copytest ~/intro
```

4.7 Keeping your disk usage down

Every file you have uses up disk space, and there is only a limited amount of total disk space for all users. On many systems, disk quotas are used to restrict the amount of space any one user can take up. To find out if you have a quota set, enter the command:

```
quota -v
```

If you have quotas set, you will be informed of your usage, quota and limit, as well as "timeleft". If

your usage is greater than your quota, you are given a small amount of time to remedy this situation before your computer account is disabled. The amount of time you are given is indicated by “timeleft”.

Even if you do not have quotas set, you should keep your disk usage down, both to ensure your directories are kept tidy and out of consideration for other users. How do you do this? Delete any files you do not need. But be careful — you will probably need your old exercises for revision at later stages. You can safely delete any files ending in `.o`. (These are files created by the compiler from your `.c` files.) You can also delete any *executable* files (e.g. files called `a.out`).

4.8 Summary

To recapitulate:

- To create a directory, use the command **mkdir**.
- To remove a directory, use the command **rmdir**.
- To view the contents of a directory, use the command **ls**.
- To move to another directory, use the command **cd**.
- To find out what directory your are currently in, use the command **pwd**.
- To move or files around or rename them, use the command **mv**.
- To find out how much disk quota you have, use the command **quota -v**.

G Submitting Your Project

1 Submit

Most of your assessable projects are to be submitted using the `submit` command.

It is used as follows:

```
submit unit_number project file(s)
```

where `unit_number` is a three-digit course code such as '171' or '172', and `project` is a name such as '1', 'B' or '2a', and `file(s)` is a (space separated) list of the files that you want to submit (in many cases this will be just one file, but it can be more).

For example, to submit a file `projA.c` for Project A in Unit 171:

```
submit 171 A projA.c
```

2 Verify

After submitting your project, you should run the `verify` command to confirm that your submission was accepted successfully. This command displays your submission on your screen, or if you choose, the output can be re-directed to a file. Thus, to verify the submission of the above example, you might enter the command:

```
verify 171 A > projA.ver
```

With this command, the verification of your submission is directed into the file `projA.ver`, which you can then examine at your leisure. Make sure you check the contents in this file. Open it with vi to see the output from verify.

Warning: Do not use, for this verification re-direction, a filename that is exactly the same as your source file (in the above example for instance, do not use the name `projA.c`, because that source file will be over-written by the verification file.)

You can verify your submission as often as you choose.

3 Re-Submissions

You can also re-submit as often as you choose. But only one submission — the most recent one — is retained by the system; any earlier submission is over-written by a subsequent submission. It is a wise precaution to make a first submission of your project at least 2–3 days before the closing date even if that submission is not your final version, in case the system goes down or becomes overloaded on the closing day; or in case your later improved version does not work! You can then submit your later versions as they become available.

4 Using External Computing Facilities

The `submit` command uses the Department's systems and programs to run your project submission. Feel free to develop your programs on your home computer, or elsewhere, but your final program must compile and run on the Department's system.

If you develop your program on a PC running Windows, make sure you convert the source files to UNIX format before submitting. On our machines the command `dos2unix` will do this conversion for you. For example, a 171 project developed on a PC running Windows (projA-dos.c) would be converted to its corresponding UNIX format and written to the second filename listed (projA-unix.c) with the following command:

```
dos2unix projA-dos.c projA-unix.c
```

Of course, both these filenames can be any name you choose them to be.

It is your responsibility to have your programs work on the Department's machines.

Be warned: even if a program runs perfectly on your home computer, it will not necessarily run correctly on the Department's computers, due to different compilers, etc. It is *your* responsibility to ensure that your program compiles and runs correctly on the University's computers *before* the project due date.

H Working At Home

Instructions on various ways to connect to the CSSE servers from your computer at home are provided in the CSSE systems documentation at `http://www.csse.unimelb.edu.au/local/tech/services/sysadmin/remote/`

I More About UNIX

The previous section of this manual served as a brief introduction to UNIX, providing enough to get you started in 433-171/151. This section contains some slightly more advanced topics in UNIX which you need in first year, some in 433-171/151, and others in 433-172/152. Refer to the *Schedule of Unix commands* for details of which commands you should know at each stage of your course.

Of course, there is a *lot* more to UNIX than what is contained in this manual. If you continue studying Computer Science in later years, you will learn more there, but you may also wish to find a good reference book on UNIX to learn on your own. (There are a couple of references given at the start of the previous section, but there are many more books available. Most large book stores will have a number of such books, and it is worth taking time to browse and find one that you prefer.)

1 Piping and Redirection

In previous sections, you have used the piping (`|`) and redirection (`>`) *operators*. (For example: `man -k copy | more`; `cal 3 2003 > Mar2003`.)

What exactly do these operators do?

- The redirection `>` operator takes the output of a command and writes it to a file.
- There is another redirection operator (`<`) which directs the contents of a file to be the *input* to a command.
- Another related operator is the concatenation (`>>`) operator, which takes the output of a command and adds it to the *end* of a file (whereas the redirection operator will overwrite the old contents of a file).
- The piping operator takes the output of a command, and uses it as the input for another command. For example, the using the command `w | more` is the same as using the following series of commands:

```
w > tmp
more tmp
rm tmp
```

You may find these operators very useful when you are testing some of your C programs.

2 UNIX Command Options

In a previous section, we looked at the command `rm -i *`. This is an example of a command used with an option. Another example of a command we have seen used with an option is `man -k`.

Most commands have options, indeed most have several options. The options available for any command are described in the `man` page for that command. Let's look at a few of the options for `ls`. Enter the commands:

```
ls -l
ls -F
ls -s
ls -Fls
```

(Use the alphabetic character `l`, not the number `1` in these commands.)

See if you can work out what each of the options does. Check the manual page to confirm your guesses.

Check the man pages of other commands you commonly use (e.g. `cp`) to see what options they can take.

3 Keeping a Backup Copy

When you are doing important work (e.g. projects), you should always keep a backup copy of your work. Some of your projects will have several stages, the idea being you complete one stage and then go on to the next. After each stage, you should make a backup copy. This way, if you run into difficulties in the next stage, you can go back to your previous work.

Let's say you are working on your first project, and you've just finished stage one. Change into the directory that contains the work for this project. Say you've called the file with the first stage your project `projA-1.hs` (or `projA-1.c`). First of all, we need to create a directory where we'll store your backup copies (if you have not already done so). Enter the command:

```
mkdir ../saved
```

This will create a subdirectory (in the directory above this one) called saved. You can use this directory for all your backups, not just the ones for this project. Now enter the commands:

```
cp projA-1.hs ../saved
cp projA-1.hs projA-2.hs
```

(or

```
cp projA-1.c ../saved
cp projA-1.c projA-2.c
```

)

This means that if something goes wrong, you'll have a backup of all the work up to stage 1. (You should do a similar backup after each stage.)

4 Functions; .bashrc and .bash_profile files

UNIX allows you to give simple names to complex commands to make typing faster. This is done using the `function` command. Enter the following:

```
function ll() { ls -l; }
```

(If it gives an error, or when you pressed return you got a line starting with '>', you probably did not include the whitespace — this is one of the few times that the whitespace (after the open brace and

before the close brace) is important.) Now enter the command `ll` and see how it behaves like the old command `ls -l`.

Now enter the command:

```
ls -a
```

This time you should see some further files in your current directory. The names of these files begin with the `.` character, and may include two files named `.bashrc` and `.bash_profile`. These two files are used to customise the behaviour of your account.

If you do not have these files (their names will not appear in the listing if you do not) then you will need to copy the default system ones into your home directory in order to be able to customise your account to your liking. The system files contain reasonable defaults, so you could get by without your own copies, but as time goes by you will find that customisation makes working with UNIX much more agreeable. Ask your demonstrator or classmates for the locations of the system files if necessary.

The `.bash_profile` file is important because the commands it contains are executed every time you log on. Usually, there is a line at the bottom of your `bash_profile` which looks like this:

```
source  /.bashrc
```

This means that every time you log on, the commands in your `.bashrc` will *also* be executed.

Take a look at your `.bash_profile` and `.bashrc` files by using `more` or `less`. Any functions that have already been set up for you will probably be in your `.bashrc`.

Below are some functions that other users have found useful. Examine these, determine their effects (by testing them and by using the `man` command), and decide whether or not you would like to include these, or others, in your `.bashrc` file.

```
function cp() { /bin/cp -i; }
function mv() { /bin/mv -i; }
function rm() { /bin/rm -i; }
function l()  { less; }
function ls() { ls -aF; }
function ll() { ls -alsF; }
function m()  { more; }
function moer() { more; }
```

If you do modify your `.bashrc` file and then want to see the effects of your modifications, first enter the command

```
source .bashrc
```

in order to activate your new `.bashrc` file. Note that functions that you create at your UNIX prompt will only exist for your current session, whereas functions that you put in your `.bashrc` file will be there every time you log in.

5 History List and Recalling Commands

Enter the command:

```
history
```

to display a listing of the most recent UNIX commands that you have made. You may wish to pipe

the output from history into `more` if it fills more than one screen (i.e. `history | more`).

Whilst this list may be of interest in its own right, its greater value and interest for you will lie in the use you can make of this to re-execute a previously-executed command. You will need to experiment using your own history of commands, to see exactly what occurs in various instances. The following example illustrates some usages.

Suppose that you have the following history list:

```
60 pwd
61 cd
62 vi .bashrc
63 source .bashrc
```

The current event is therefore number 64.

Suppose you now want to execute a command that is the same as event number 62 (that is, make another change to your .bashrc). You could enter any one of the following:

```
!62      !-2      !vi
```

The first of these re-executes the numbered command, No. 62; the second re-executes the command that is 2 before the current number; and the third re-executes the most recent command that commences with `vi`.

To execute a command similar to number 62 but on `.bash_profile` instead of `.bashrc`, we can use the *substitute* command:

```
!62:s/bashrc/bash_profile/
```

This would then execute the command: `vi .bash_profile`.

But, you must experiment using your own history of commands, and learn how this command re-call provision can assist you greatly, especially if you use long command calls.

6 Finding out Who else is On-line

There are times when you will find it useful to know who else is logged on, and how much the machines are being used. There are a number of UNIX commands which allow you to do just that:

```
who (or who | more)
w (or w | more)
finger (or finger | more)
```

Try them for yourself, you might want to use them later. Also, the last two commands can take a username as an optional argument. For example, if your friend's username was "fred", you could find out if he was logged in to your machine by using the command `w fred` or `finger fred`. Try this out. Notice that the `finger` command gives very different output depending on how you use it.

7 Combining Commands into one Command Line

Do you want to create a new subdirectory into which you will place backup copies of your Haskell program script files?

The following single command line uses the semi-colon (`;`) to separate the line into three separate commands that are executed sequentially.

```
mkdir saved ; cp *.hs saved ; ls saved
```

Make sure you understand the three commands that this command line executes, then try a similar multi-command line of your own choosing.

8 Summary

To recapitulate the major points of interest in this section:

- To modify the behaviour of a command, use an appropriate command option. Command options generally follow the name of the command and consist of a '-' and a character.
- Functions are a way of conveniently associating a number of commands with a name which you supply. Put the functions you wish to keep into your .bashrc file.
- To redirect the output of a command into a file, use the $>$ operator.
- To redirect the output of a command into the input of another command, use the $|$ operator.
- To view your command history, use the command **history**.
- To run a command in the command history, use the **!** operator.
- To combine commands into one line, separate them by the **;** operator.
- To find out who else is logged on, use the following commands: **who**, **w**, **finger**.

J Introduction to vi

The UNIX command `cat` can be used to create and write a file, but it is too limited a tool for all except the simplest cases. Usually you will use a text editor which has an extensive range of commands to create and modify files. UNIX has several text editors (such as `ed, sed, vi, emacs`). These notes refer to the visual editor, `vi`.

If there are so many other editors available, why do we insist that you learn this complicated one called `vi`? Well, `vi` is a very powerful editor, when you become experienced, and unlike some other popular editors, it is available in nearly identical form on nearly every UNIX system. This means that once you have mastered this editor, you will be able to create and modify files whatever UNIX machine you are using.

`vi` is called a *visual editor* because you can see a copy of the file you are creating or modifying on the screen while you are editing. (Yes, files were once edited without being able to see the file.) `vi` is a powerful and widely-used editor. Here, you will be introduced to only a few of its many commands and capacities.

The `vi` editor has two distinct modes — *command* mode (with its own set of commands), and *insert* mode. In command mode, most commands (key strokes) either move the cursor around the editor buffer, or enable you to alter the text (change, move or delete it) in the editor buffer. Other commands (including `i, a, o`) cause the editor to change to *insert* mode. All characters typed in insert mode are added to the file. Typing the ESC key causes the editor to leave insert mode and return to command mode.

You should also know that when you enter the command: `vi filename`, a *copy* of that file `filename` is placed into an *editor buffer*, and the editing that you do is performed on this copy, not on the original file. The original file is only replaced by the changed text when you *write* (save) the editor buffer using the `:w` (or `ZZ`) command. If `filename` is a new file, the editor buffer is empty until you insert text, and the file is not created until you save or write the buffer.

Occasionally you will make such a mess with your editing that you will want to revert to the original without saving the alterations. the command `:q!` throws away the text in the editor buffer and leaves the original file unaltered. Alternatively you could use the command `:wq filename2` and write your edited file to a new filename, `filename2`, leaving the original file unaltered (i.e. not overwritten).

Here are two further items that you need to keep clearly in mind, to get you out of trouble:

- If you find strange and extraneous characters on your screen, or if the display is somehow distorted, it is possible that you have you have not set your terminal type correctly for the terminal you are currently using. To correct this, you need to exit from `vi` (type ESC to exit from insert mode, then `:q!` to quit without saving), set your `term` variable correctly, and then resume editing. (Refer to the section *UNIX Troubleshooting* to see how to do this.)
- If you become confused as to what mode you are in (*command* mode or *insert* mode) while using `vi`, e.g. if it does not execute your commands correctly, press the ESC key to return to command mode. It is always safe to press the ESC key; it may sometimes be unsafe to press other keys.

1 vitutor

The program `vitutor` copies a text file on to your directory working area, and provides a relatively short introduction to `vi`. You should now access and proceed through this program, by entering the command

```
vitutor
```

Basic vi Commands (for absolute beginners)

h, j, k, l	Move one character left/down/up/right
x	Delete the character currently under the cursor
i, a	Insert characters before/after cursor
:q!	Quit without writing the editor buffer
:wq or ZZ	Write editor buffer (overwriting original file), then quit
ESC	Exit from insert mode and return to command mode

More Basic vi Commands

*n*SPACE, *n*BACKSPACE	Move *n* characters forward/backward
*n*RETURN, *n*-	Move to start of *n*'th next/preceding line
*n*h, *n*j, *n*k, *n*l	Move *n* characters left/down/up/right
*n*w, *n*b	Move *n* words forward/backward
*n*G, (G)	Move to line number *n* (default: last line)
`^F, ^B`	Scroll one screen forwards/backwards in your file
*n*dl, *n*dw, *n*dd	Delete *n* chars/words/lines
*n*x	Delete *n* characters (cf. *n*dl)
I, A	Insert characters at beginning/end of line
o, O	Open line for insertion below/above cursor

When you have finished the `vi` tutorial program, you should read again the brief description of `vi` given above; it should be more meaningful to you now. You might also like to go through the tutorial program again, to help consolidate your new learning.

2 Summary

Once you have completed this section, including going through `vitutor`, you should be able to commence using the `vi` editor to create and modify your files.

- Do *not* expect to remember all the commands you covered in `vitutor`. Remember the basic ones given in the previous two tables to start with, and remember the sorts of things that *can* be done.
- If you find you need to do something, but cannot remember how, refer to the table at the back of this manual, or look at the file created by `vitutor`, `tutor.vi`, in your home directory. This file contains all the text of the tutor program, so you can look up anything in there.

- Some of the more advanced features of `vi` are discussed in the next section of the manual. It is not necessary to cover these topics in first year Computer Science, but you may find them useful anyway.

K More about vi

This section of the manual discusses some of the more advanced features of `vi`. In first year, it is not necessary to know these features, but you may find some of them useful, so it will be well worth your while to read through this section at least once.

1 High-level Commands

Writing, quitting, editing new files

NOTE: any command starting with a colon (`:`) or slash (`/`) must be completed by pressing ENTER or RETURN. For other commands you do not need to press either of these keys, unless otherwise specified.

So far we have seen how to enter `vi` and to write out a file using either `ZZ` or `:w`. The first exits from `vi`, (writing if changes were made), the second writes and stays in `vi`.

If you have changed `vi`'s copy of the file, but do not wish to save your changes, either because you messed up the file or decided that the changes are not an improvement, then you can give the command `:q!` to quit from `vi` without writing the changes. You can also re-edit the same file (starting over) by giving the command `:e!`. These commands should be used only rarely, and with caution, as it is not possible to recover the changes you have made after you discard them in this manner.

You can edit a different file without leaving `vi` by giving the command `:e` *name*. If you have not written out your file before you try to do this, then `vi` will tell you this, and delay editing the other file. You can then use `:w` to save your work, and give `:e` *name* again, or carefully give the command `:e!` *name*, which edits the other file and discards the changes you have made to the current file.

Escaping to a shell

You can get to a shell to execute a single command by giving a `vi` command of the form `:!` *cmd*. The system will run the single command `cmd`. When the command finishes, `vi` will ask you to "hit return to continue". When you have finished looking at the output on the screen, you should hit ENTER or RETURN and `vi` will redraw the screen. You can then continue editing. You can also give another `:` command when it asks you for a return; in this case the screen will not be redrawn.

If you wish to execute more than one command in the shell, then you can use the command `:sh`. This will give you a new shell. When you terminate the shell, by typing a `^D`, `vi` will clear the screen and continue.

If `vi` has been invoked from `csh`, `^Z` will suspend `vi` and return to the (top level) shell. When `vi` is resumed, the screen will be redrawn.

Marking and returning

The command `''` (two back quotes) returns to the previous place after a motion of the cursor by a command such as `/`, `?` or `G`. You can also mark lines in the file with single letter tags and return to these marks later by naming the tags. Try marking the current line with the command `m` *x*, where you should pick some letter for *x*, say 'a'. Then move the cursor to a different line and hit `'a`. The cursor will return to the place which you marked. Marks will last only until you edit another file.

When using operators such as `d` and referring to marked lines, it is often desirable to delete whole lines rather than deleting to the exact position in the line marked by `m`. In this case, you can use the form `'` *x* rather than `‘` *x*. Used without an operator, `'` *x* will move to the first non-white character of the marked line; similarly, `''` moves to the first non-white character of the line containing the previous context mark `‘‘`.

Adjusting the screen

If the screen image is messed up because of a transmission error to your terminal, or because some program other than `vi` wrote output to your terminal, you can hit a `^L` to refresh the screen.

On a dumb terminal, if there are ‘@’ lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing `^R` to cause `vi` to retype the screen, closing up these holes.

Finally, if you wish to place a certain line on the screen at the top, middle or bottom of the screen, you can position the cursor to that line, and give a `z` command. You should follow the `z` command with ENTER or RETURN if you want the line to appear at the top of the window, with `.` if you want it at the centre, or with `--` if you want it at the bottom.

2 Special Topics

Recovering lost lines

You might have a serious problem if you delete a number of lines and then regret that they were deleted. Despair not, `vi` saves the last 9 deleted blocks of text in a set of registers numbered 1–9. You can get the *n*’th previous deleted text back in your file with the command ” *n*`p`. The ” says that a buffer name is to follow, *n* is the number of the buffer you wish to try (use the number 1 for now), and `p` is the print command, which puts text in the buffer after the cursor. If this does not bring back the text you wanted, hit `u` to undo this and `.` (period) to repeat the put command. In general, the `.` command will repeat the last change you made.

As a special case, when the last command refers to a numbered text buffer, the `.` command increments the number of the buffer before repeating the command. Thus a sequence of the form ”`1pu.u.u.` will, if repeated long enough, show you all the deleted text which has been saved for you. You can omit the `u` commands here to gather up all the text in the buffer, or stop after any `.` command to keep just the then-recovered text. The command `P` can also be used rather than `p` to put the recovered text before instead of after the cursor.

Recovering lost files

If the system crashes, you can recover the work you were doing to within a few changes. You will normally receive mail when you next login, giving you the name of the file which has been saved for you. You should then change to the directory you were using when the system crashed and give the command

```
vi -r name
```

replacing *name* with the name of the file which you were editing. This will recover your work to a point near where you left off.

Do NOT *use* `ZZ` *to write the recovered file unless you are sure you have changed it since it was recovered, or* `vi` *will simply discard the file.*

You can get a listing of the files which are saved for you with the command

```
vi -r
```

If there is more than one instance of a particular file saved, `vi` gives you the newest instance each time you recover it. You can thus get an older saved copy back by first recovering the newer copies.

Continuous text input

When you are typing in large amounts of text, it is convenient to have lines broken near the right margin automatically. You can cause this to happen by giving the command

```
:set wm=10
```

This causes all lines to be broken at a space at least 10 columns from the right hand edge of the screen.

If `vi` breaks an input line and you wish to put it back together you can join lines with `J`. You can give `J` a count of the number of lines to be joined, as in `3J`, to join 3 lines. `vi` supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. You can kill the white space with `x` if you do not want it.

Features for editing programs

`vi` has a number of commands for editing programs. The thing that most distinguishes editing of programs from editing of text is the desirability of maintaining an indented structure in the body of the program. `vi` has an *autoindent* facility for helping to generate correctly indented programs. To enable this facility give the command

```
:set ai
```

Now try opening a new line with `o` and type some characters on the line after a few tabs. If you now start another line, `vi` supplies white space at the beginning of the line which lines it up with the previous line. You cannot backspace over this indentation, but you can use `^D` to backtab over the supplied indentation.

Each time you type `^D` you back up one position, normally to an 8 column boundary. This amount is settable; `vi` has an option called `shiftwidth` which you can set to change this value. Try giving the command

```
:set sw=4
```

and then experimenting with autoindent again.

For shifting lines in the program left and right, there are the operators < and >. These shift the lines you specify right or left by one `shiftwidth`. Try << and >> which shift one line left or right, and <`L` and >`L`, which shift the rest of the display left and right.

If you have a complicated expression and wish to see how the parentheses match, put the cursor at a left or right parenthesis and hit `%`. This will show you the matching parenthesis. This works also for braces '{' and '}', and brackets '[' and ']'.

If you are editing C programs, you can use the `[[` and `]]` keys to advance or retreat to a line starting with a '{', that is, a block at a time. When `]]` is used with an operator it stops after a line which starts

with '}'; this is useful with `y]]`.

3 vi and ex

`vi` is actually one mode of editing within the editor `ex`. When you are running `vi` you can escape to the line oriented editor of `ex` by giving the command `Q`. All of the `:` commands which were introduced above are available in `ex`. Just give them without the `:`, and follow them with a ENTER or RETURN. Likewise, most `ex` commands can be invoked from `vi` using `:`.

In rare instances, an internal error may occur in `vi`. If this happens, you will get a diagnostic and be left in the command mode of `ex`. You can then save your work and quit, by giving the command `x` after the `ex` prompt `:`, or you can re-enter `vi` by giving `ex` the command `vi`.

There are a number of things which you can do more easily in `ex` than in `vi`. Systematic changes in line-oriented material are in this category. The section below gives an introduction to some simple `ex` commands[2]. Experienced users often mix their use of `ex` command mode and `vi` command mode to speed the work they are doing.

ex commands used in vi

This is a brief introduction to some of the `ex` commands often used within `vi`.

Since `ex` is a line oriented editor, the commands given must specify which line or lines to affect. Lines are specified in the following ways:

Current line The current line is referred to as line dot, ie. '`.`', and is officially the last line you did anything to. In `vi` this is the line the cursor is on.

Line numbers In `vi` the command `^G` can be used to find out the line number of the current line.

Last line When specifying line numbers the symbol `$` refers to the last line in the buffer.

Context search Lines can be specified by giving a pattern to search for. The first line that matches the given string is the line on which the command is applied.

Where commands are shown with two lines specified, the operation is applied from the first to the second inclusively. However it is not necessary to always specify two lines. If one line is given the comma is omitted. If no line is specified, the current line is assumed.

Print

The `ex` print command is not very useful in `vi`, but it is given here because it is a harmless command to practice line references with. The print command has the following format:

```
:line1,line2 p
```

Some examples are:

[2] If you need more information refer to the **Advanced Editing on UNIX** or the **Ex Reference Manual** documents in the **UNIX User's Manual**. The **Advanced Editing on UNIX** document concerns itself with the editor `ed`, but all of the commands discussed apply exactly to `ex`.

```
:.,.+2p
```

which prints the current and next 2 lines;

```
:10,$-1p
```

which prints line 10 down to the second last line of the file; and

```
:/the/,/and/p
```

which prints all the lines between the first occurrence of the word ‘the’ and the first occurrence of the word ‘and’.

Substitute

The format is:

```
:line1,line2 s/string1/string2/
```

The effect is that wherever string1 is found between the lines specified, string2 is substituted for it. String1 can be a regular expression. For example:

```
:s/er/or/
```

changes er to or on the current line. Also

```
:s/x.y/xy/
```

gets rid of the character between x and y irrespective or what character was present.

Note that if string1 is not found nothing happens but a warning message appears on the bottom of the screen.

In string2 there are only two special characters:

\ (backslash) and & (ampersand)

The backslash \ has the same effect as in string1. Ampersand & in string2 refers to the string that was matched by string1. For example:

```
:s/computer/&s/
```

pluralises the word ‘computer’ with minimum typing, and

```
:s/.*/(&)/
```

puts parentheses around the whole line.

All of the other regular expression special characters have no extra-ordinary affect in string2. For example:

```
:s/x.y/x.y/
```

replaces any character found between x and y with a literal dot.

Global substitute

In the above examples given for the substitute command, the normal `vi` mode commands would have been just as effective in making the changes. However the substitute command becomes much faster than using `vi` commands when repeated changes over several lines, or the whole file have to be made.

For example, if you misspelt someone's name several times in the first ten lines of a letter, the command

```
:1,10s/Lyn/&ne/
```

will change any 'Lyn's found on the first ten lines to 'Lynne'.

However if more that one occurrence of 'Lyn' is found per line, only the first occurrence will be changed. To get around this put a `g` on the end of the command:

```
:1,10s/Lyn/&ne/g
```

To specify that you want to make the change in all the lines of the file you could use

```
:1,$s/Lyn/&ne/g
```

or

```
:g/Lyn/s/Lyn/&ne/g
```

The second one says to globally search for the string 'Lyn' and where it is found apply the substitution. This can also be written as:

```
:g/Lyn/s//&ne/g
```

The empty string1 after the *s* indicates that it is the same as the string that was just searched for.

When complex substitutions are needed it is often good to quickly check the lines that were changed to make sure that it did what you thought it would. The print command can be combined with the substitute command to display all of the affected lines. Just put a `p` on the end of the command. That is:

```
:g/Lyn/s//&ne/gp
```

Delete

The delete command has the format:

```
:line1,line2 d
```

This command is useful in `vi` if you want to delete more than one screen full of text. For example:

```
:.,$d
```

will delete from the current line to the end of the file in one step.

Write

The write command `w` was introduced in the section "Writing, quitting and editing new files", but this command can also take line numbers and file names. For example:

```
:1,/^Dear/-1 w > heading
```

will make a copy of the heading section of a letter in the file called heading. The line /^Dear/-1 is the line before the one beginning with 'Dear'.

Read

The read command is used to read a file into the buffer. If there is a file in the buffer already the effect is to add the text of the new file to what is already there. The text of the new file is placed after the current line. This command is used by positioning the cursor at the appropriate position and issuing the command:

`:r` *filename*

L Electronic Communication and the World Wide Web

Your CSSE UNIX account includes access to the Internet, including electronic mail, news and the Web.

Warning: You are reminded as per the University's rules regarding the use of University computing facilities (which are contained in your student diary), your access to university computing facilities is strictly for academic purposes only. Your use may be monitored by the department, and access is partially restricted.

1 Electronic Mail (email)

There are a number of programs available which enable you to send and receive electronic mail, but the one that we recommend for new users is **pine**.

To use this program, simply type `pine` at your UNIX prompt. Pine has extensive on-line help, usually obtainable by pressing the `?` key. You will probably be able to go ahead and use it now without any further instructions, but we will briefly discuss the basics of sending and receiving email below.

When you want to quit **pine**, in most cases you just need to press `Q` — if this does not work, check the options at the bottom of the screen.

1.1 Your email address

Your email address is the way in which people can contact you from all over the world via the Internet. It is totally unique — this ensures that no matter who sends the email, if they type in your address correctly, it will get it you. For people on the same system as you (i.e. if you have a CSSE account, and they are sending mail from their own CSSE account), your email address is simply your login name. Other people must use the long form of your email address. If your login name was *foo* and you have a CSSE account, your full email address would be *foo@students.csse.unimelb.edu.au*.

If you want people to be able to contact you via email, they must know this address. (If you send them email first, your address will automatically be contained in that email.)

1.2 Receiving email

When you start **pine**, if you have received any new email since you last read your mail, it will be in the "INBOX" folder, which will always be the current folder when you start the program. So to check for new mail, choose `I` from the main menu.

If you do have any new mail messages, they will be listed on your screen, showing the sender and the subject of the message, amongst other things. Pressing the RETURN or ENTER key will display the current message. You can also use `N` to read the next message, or `P` to read the previous message.

After you have read a message, you might like to reply to it — see the section below on sending email for more details of how to do this. When you have finished reading and replying to a message, you should either delete it or save it (`D` to delete and `S` to save).

1.3 Sending email

If you wish to send email, you can either reply to a message that you have received (using `R` when you are viewing the message), or create a new message to send by choosing `C` from the main menu. If you reply to a message, the program will automatically choose the correct return address for your message, but if you are creating a new message, you will need to know the address of the person you want to send the email to.

After `To`, you should have the address of the person you wish to send the email to. This can actually be more than one person — in this case, the addresses must be comma-separated.

`Cc` stands for carbon copy — if you include an address here, the message will also be sent to this person. Usually you would leave this line blank.

`Attchmnt` is used if you wish to send a file to someone. You will notice when you have your cursor on this line that `^T` is an option. This option will list your files — you then move the cursor to the file you want to send and press return. You should leave this line blank if you do not wish to send any files.

The next header line is `Subject`. This should be the subject of your message. If you are replying to a message, it will automatically fill in the subject for you, based on the subject of the message you are replying to. You can still change it if you wish.

After filling in these lines, you should fill in the message text using the basic editor provided by **pine**. (You may find this editor, which is called **pico** easier to use than **vi**, but it is not nearly as powerful.) When you have completed your message, you can send the message by pressing `^X`. Alternatively, if you change your mind about sending the message, you can choose `^C` to cancel your message.

Once you have sent your message, you cannot cancel it.

The authoritative documentation on the mail system is in the department's systems documentation at `https://www.csse.unimelb.edu.au/local/tech/services/sysadmin/mail/`. This includes instructions on how to forward your CSSE mail to another address (for example, if you already have a hotmail account and prefer to read your mail there).

It is important that you read your CSSE email, as it will be used by lecturers, tutors and the system administrators to contact you.

2 Electronic News

Electronic news is a collection of discussion forums. As with electronic mail, there are many news readers available for electronic news, however we recommend that new users read news with **pine**.

To use this program, simply type `pine` at your UNIX prompt. To access the newsgroups, press `M` to get to the main menu, then press `L` to get the folder list. You can select from two sources of news: `CSSE-News` contains groups specific to the department, including a news groups for each subject; and `UniMelb-News` to access external news groups.

There are literally thousands of news groups on just about every topic under the sun. If you wish, you can plough through every newsgroup and decide whether or not you wish to read each one. Be warned though that there are nearly 3000 newsgroups, so this will take a *long* time. If you decide that you do *not* want to read a newsgroup, type `D` (capital 'd') to *unsubscribe* to a newsgroup — you will be asked for confirmation each time you do this.

Posting a news item to the group is the same as sending email. Select the news group in the list and press `C` to start composing a message.

The authoritative documentation on the operation of the news system is in the department's systems documentation at `https://www.csse.unimelb.edu.au/local/tech/services/sysadmin/news/`

3 The World Wide Web (WWW)

WWW pages are *hypertext* documents published on the Internet, consisting of graphics, text and links to other documents. Document locations on WWW are specified by *URLs* (Uniform Resource Locators). These URLs will look something like this: `http://www.csse.unimelb.edu.au` (this is the URL for the CSSE homepage).

You should learn how to access the WWW while at CSSE as much of your course material and documentation on using the computer facilities is published there on the department homepage `http://www.csse.unimelb.edu.au/`.

Again, there are several document browsers available for the WWW. We recommend **mozilla** for X-terminals, **firefox** for Windows PCs, and **lynx** from text-based UNIX shells or via a modem.

Access to the Web from the CSSE laboratories and UNIX servers is via the department *Web Proxy*. Set your browser automatically download its proxy caching configuration from `http://www.csse.unimelb.edu.au/proxy.pac`

The authoritative documentation on the web access is at `https://www.csse.unimelb.edu.au/local/tech/services/sysadmin/web/`.

3.1 Mozilla

Mozilla is a web browser for X-windows. To start up mozilla, enter the command

```
mozilla &
```

at the UNIX shell prompt.

3.2 Lynx

Lynx is a text-based web browser. Most pages which contain graphics also contain text alternatives, so unless you are looking for pictures, lynx should be fine for your purposes most of the time. Unfortunately there are still a few pages around which are entirely graphics-based — this is *not* a problem with lynx, but a problem with the designer of the page. There should *always* be a text alternative to graphics, and this is something you should bear in mind if you come to be writing pages yourself.

To start lynx, enter the command `lynx` at your UNIX prompt. As with mozilla, it will start up by default with the departmental homepage. Again, as with mozilla, you can also specify a URL in the command line to get it to start with a different page (e.g.

```
lynx http://www.unimelb.edu.au
```

will make it start with the University homepage instead).

To follow links using lynx, use the up and down arrow keys to move to the highlighted words (even if the words are on the same line, you should still use the up/down keys, not left/right). When you reach

the link that you are interested in, use the right arrow key to follow the link. If you wish to go back to the previous document, use the left arrow.

If you have already started lynx and wish to go to a specific URL, use the *goto* command by pressing `g`. You will then be prompted

```
URL to open:
```

which is when you should type in the URL you wish to visit.

If you are searching for information on a particular topic, you can also do a net search using lynx. In lynx, use the *goto* command (mentioned above) to go to URL `http://www.infoseek.com`. Use the down arrow to move to the *text entry field* (below "Helpful Tips") and enter the keywords you wish to search for. When you have entered the words, use the TAB key to move to "Search", and then the right arrow to start your search. After a short time, a list of links matching your keywords will be generated.

The exit the lynx program, simply type `Q`.

4 File Transfer – `sftp`

There may sometimes be a need for you to transfer files between systems. In the chapters in UNIX you will have become familiar with copying and moving files with the `cp` and `mv` commands. If, however, you wish to copy a file on one machine (eg on an ECR machine or any other Internet connected machine), to your account at CSSE, then you will need to use the `sftp` program. **Note that `sftp` is *not the same as the more common FTP facility present in web browsers.* `sftp` encrypts your login and password so that it cannot be easily seen by someone monitoring your Internet connection.**

In the following, we will refer to the *local* machine as the machine that you are starting on, and the *remote* machine as the machine that you wish to connect to. We describe the interactions for a machine operating under UNIX however `sftp` or its equivalent is available for the most common operating systems such as MS-Windows, and MacOS (including OS/X).

To use it, simply type `ftp` at the UNIX prompt followed by the name of the machine you wish to connect to, e.g. `sftp lister.csse.unimelb.edu.au`. Enter your login name when prompted, and then your password for the *remote* machine, not the local one.

Next you will see the `sftp>` prompt. This happens whether your password was typed correctly or not, although if you get it wrong, there will a message indicating this. If you got your password wrong, you will not be able to transfer any files, so you must try again. To do this, quit out of the `sftp` program by typing `quit`, and start the process over again.

Once you have successfully connected, there are a number of important commands that can use from the `sftp>` prompt:

- `help`. Gives you a list of all the commands!
- `get`. This command allows you to transfer a file from the remote machine to the local machine.
- `put`. This allows transfer from the local machine to the remote machine.
- `ls, cd` work the same as in UNIX, but act on the remote machine.
- `!`. You can access local UNIX commands using the `!` character. This is often used with the `ls` command (i.e. `!ls`) to check the files on the local machine.
- `lcd`. This changes directories on the local machine. You may have already discovered that `!cd` does not work (because of the way UNIX works)!
- `ascii, binary`. These commands tell `sftp` what kind of file you are transferring. When both systems are UNIX (as will most often be the case), then there will be no difference. If you are transferring between UNIX and MS-DOS, however, then the ascii mode will make some small changes to the file during transfer. Normally you will be transferring only text files, for which you should use the `ascii` mode.

To illustrate the use of `sftp` let us consider the following example: Here we are starting at gromit (an ECR machine but this could also be any Internet connected machine with `sftp` installed) and are connecting to lister (a CSSE machine) So lister is the remote machine and gromit is the local one. Italic type (*`like this`*) is used to indicate what you type, while normal typewriter text shows you the output from the computer.

```
gromit% sftp lister.csse.unimelb.edu.au
jsmith@lister.csse.unimelb.edu.au's password: here you type your password
sftp>
```

We are now in the `sftp` program and can transfer files. Let's say we want to copy a file from CSSE to ECR. It's a text file, so we first set the mode to A (ASCII). We then use the `get` command.

```
sftp> ascii
File transfer mode is now ascii
sftp> get project.hs
sftp> get simple.hs
simple.hs        |          0 kB |   0.2 kB/s | TOC: 00:00:01 | 100%
sftp>
```

We can also send a file from ECR to CSSE with the `put` command: (and then quit the program).

```
ftp> put exercise1.c
exercise1.c      |         19 kB |  10.9 kB/s | TOC: 00:00:02 | 100%
ftp> quit
gromit%
```

As mentioned above, the ECR machine gromit is just an example here. This could also be your machine at home.

5 Other Forms of Electronic Communication

As well as mail and news, there are two other methods of electronic communication which are sometimes useful. These are the `talk` and `write` commands.

Never send `talk` or `write` messages to people that you do not know. Also, do not send messages to staff members using these commands, unless they have explicitly instructed you to do so. The most likely result if you do is that your message will be ignored.

Please be aware that there are laws regarding electronic harassment of people. If you are discovered to be harassing anyone via the computer, such as by repeatedly sending unwanted messages, you may have your account suspended. If you believe you are being harassed, you should feel free to discuss this with staff (also, see the section below on how to refuse messages).

5.1 write

`write` is used to send a *brief* message to someone who is logged on to the same machine as you. To send a message to someone with login name *foo*, you would enter the following command:

```
write foo
```

(But remember, *foo* must be logged on to the same machine as you. If he/she is not, you will get a message saying "`foo is not logged on`".)

After you have entered this command, everything you enter will be printed on *foo*'s screen. This can be very annoying for *foo* if he/she is trying to edit a program or do some other sort of work (because the message will appear in the middle of whatever they are doing), so please keep your message brief (no more than 5 lines or so). When you have finished your message, press ENTER and then `^D`. The letters `EOT` (end of transmission) will then appear on *foo*'s terminal.

5.2 talk

The `write` command is useful for some things, such as sending a reminder, but for interactive communication it is very cumbersome. (If two people are writing to each other at once, they can mess up each other's screens so that neither can follow what is going on.) If you do wish to have an interactive conversation with another user, you should use the `talk` command instead. This command has the added advantage that you do not have to be logged on to the same machine as the person you wish to communicate with (but you do need to know which machine they are logged on to).

To talk to someone with user name *foo* who is logged on to a machine called *gromit.ecr.unimelb.edu.au*, use the following command:

```
talk foo@gromit.ecr.unimelb.edu.au
```

After you have entered the command, your screen will be cleared, and a horizontal line will appear across the centre of the screen. At the top of the screen will be the message "`Waiting for your party to respond`".

Say your user name is *bar*, and you are logged on to *fido*. On *foo*'s terminal this message will appear:

```
Message from Talk_Daemon@fido at 14:28 ...
talk:  connection requested by bar@fido.ecr.unimelb.edu.au.
talk:  respond with:  talk bar@fido.ecr.unimelb.edu.au
```

If *foo* does wish to talk to you, he/she would enter the command

```
talk bar@fido.ecr.unimelb.edu.au
```

When they enter this command, the message at the top of your screen will change to "`Connection established`". At this stage, *foo*'s screen will look just like yours. From now on, everything you type will appear in the top half of your screen, and everything *foo* types will appear in the bottom half of your screen. When you have finished talking, press `^C` to end your conversation. The message at the top of your screen will change to "`Connection closing. Exiting`". (*foo* can also terminate the conversation in the same way — in this case you will get the same message.)

5.3 Refusing messages

Sometimes when you are working you will not want to be interrupted by messages from other people. There is a simple command to stop people bothering you:

```
mesg n
```

When you enter this command, if anyone tries to talk or write to you, they will receive the message "`Your party is refusing messages`". You can enter this command at any time, as soon as you log in, or after someone sends you a talk request, or whatever. Once you have finished whatever it was that required your concentration, you may wish to receive messages again. You can do this by entering the command

```
mesg y
```

The command `mesg` without any arguments will tell you the status of your messages (i.e. `y` or `n`). If your messages are turned off, not only will you not be able to receive messages, you will also not be able to send messages.

Turning your messages off has no affect on your ability to send or receive email.

6 Summary

To summarise what has been covered in this section:

- To send and receive electronic mail, use the command `pine`.
- To read electronic news, use the command `pine`.
- `mozilla` is the preferred WWW browser if you are using an X-terminal.
- `lynx` is the WWW browser to use if you are using an ascii terminal (including a connection via modem).
- To send a brief message (no more than a few lines long), use the command `write`
- To have an interactive conversation with another user, use the command `talk`
- To prevent people from interrupting your work with messages, use the command `mesg n`, and to resume receiving messages, use the command `mesg y`.

M Program Readability

Developing good programming style and internal documentation is essential if your code is going to be readable. Whether you are writing a program in industry, completing a project in your course, or just completing the exercises that have been set for labs or tutes, it is equally important for your code to be readable.

If you are working on a software project in industry, it is likely that you will be one of a team of programmers, and so in order to work efficiently together, you must all write code that all the other team members can read and understand. Even if you're not working in a team, most software has to be modified and/or upgraded at some stage, and this is often done by someone other than the original author. If *you* were the one having to make the modifications, you'd like it to be as easy as possible — by ensuring that your code is readable, you will be making someone else's job easier in the future.

If you are writing code for an assessed project, the readability of your project will be one of the factors which influences your mark for the project. This follows directly from the above — we want to ensure that you develop the skills *now* that you will need when you enter the workforce.

Even if you are just writing programs for your own personal use, you should continue to use good style and documentation. There are two reasons for this: firstly, it should become a habit; secondly, and more importantly, you will be surprised how little you remember of a program that was written two months ago when you come to revise. Even something that seemed perfectly clear when you wrote it may be difficult to understand a couple of months later.

The importance of *consistency* in both your programming style and your documentation style cannot be stressed enough. It is amazing how much a clear and consistent style for each can aid in the readability of a program. If you are working on a team project, it is a good idea to decide upon a style before you start your work, so that you do not have to go back and make changes to the style at a later stage.

Have a look at the following two sample programs, and then we will discuss some of the issues concerning program style and documentation. (When we are referring to documentation here, we mean *internal* documentation, not the hardcopy documentation you may be required to provide with some programs.)

```
{-------------------------------------------------------------------      (1)
||
||   Script  : factors.hs
||   For     : example of program readability in Haskell
||   Written : CS staff; Jan 98
||
||   This program prints out the factors of all numbers from
||   m to n inclusive, with the factors of each number on a
||   separate line.
||
||   Example Usage: putStr (display_factors 10 20)
||
-------------------------------------------------------------------}

-- display_factors m n: format output of factors_list m n so that       (2)
--      factors of each number are listed on seperate lines.

display_factors :: Int -> Int -> String                                 (4)
display_factors m n
    = unlines (map show (factors_list m n))

-- factors_list m n: returns a list of the form (value, factors of value)
--      for all values from m to n inclusive

factors_list :: Int -> Int -> [(Int ,[Int])]
factors_list m n = [(val, factors val) | val <- [m..n]]

-- factors n: returns the factors of n, including 1 and n

factors :: Int -> [Int]
factors n
    | n < 1 = []
    | otherwise = fac_acc n []
      where
        -- fac_acc checks each number from n down to 1 in turn to
        -- see if it is a factor of n - if it is, number is added
        -- to the list of factors.
      fac_acc 1 fac_list = 1:fac_list                                   (5)
      fac_acc m fac_list
        | n `mod` m == 0 = fac_acc (m-1) (m:fac_list)
        | otherwise = fac_acc (m-1) fac_list
```

```
/*                                                                      (1)
 *
 *   File    : factors.c
 *   For     : example of program readability in C
 *   Written : CS staff; Dec 95
 *
 *   This program finds the factors of all numbers between the
 *   two input values.
 */

#include <stdio.h>

void print_factors(int n);
void display_factors(int min, int max);
int main() {                                                            (3)
    int min, max;       /* range over which to find factors */          (3),(4)

    printf("Enter the minimum and maximum values: ");

    /* Check that the user enters valid min and max values */

    while ((scanf("%d%d", &min, &max) != 2) || (min > max)) {
        fprintf(stderr, "Error: expecting two INTEGER values, ");
        fprintf(stderr, "with first value <= second value.\n");
        fflush(stdin);          /* Clear input stream, ready for user's    (3)
                                   next input */
        printf("Enter the minimum and maximum values: ");
    }

    display_factors(min, max);
    return 0;
}
/*                                                                      (2)
 * display_factors: prints out the factors of all numbers between
 *      min and max, with each list of factors on a separate line.
 */
void display_factors(int min, int max) {                                (4)
    int i;

    for (i = min; i <= max; i++) {
        printf("Factors of %d: ",i);
        print_factors(i);
    }
}

/*
 * print_factors: prints out the factors of n in a comma separated list
 */
void print_factors(int n) {
    int i;

    for (i = 1; i < n; i++)
        if (n%i == 0)
            printf("%d, ",i);
    printf("%d.\n",n);
}
```

1 Notes About the Sample Programs

These sample programs are intended as examples of good program and documentation style. If you like, you can base your work on these styles, or you can follow a style set out in some other text. Whichever style you choose to follow, you should take each of the following into consideration (see references in sample program):

1. **Header documentation** should contain information about
   - The file name,
   - The purpose of the program
   - The author of the program
   - The date the program was written
   - A *brief* description of what the program does
2. **Function documentation** should be included for almost every function, except for trivial functions whose purpose is obvious from their name. Function documentation should briefly describe what the function does, and what the inputs and outputs are.
3. **In-line documentation** is used for brief explanations in the body of the code, such as describing variables when they are declared, or explaining small sections of code which may be unclear.
4. **Identifier names** (i.e. names of functions, variables, function arguments and symbolic constants) should be descriptive, but not over-long. When they consist of more than one word, use underscores (_) to separate words, or capitalise the first letter of each word.
5. **Symbolic constants** should be used in place of "magic numbers".
6. **Indentation** should be clear and consistent.
7. **Order of functions and symbolic constants** in your code should make it easy to follow. Usually you would group all symbolic constants at the top, followed by the top-level function, then second-level functions and so on.
8. Suitable **functional decomposition** can also improve the readability of your program — any sections of code which serve a logical function should be grouped together as a separate function. You should avoid letting your functions grow too long — if you cannot clearly see where a function starts and ends with a single glance, it's a sign that it should be decomposed into smaller functions.
9. One of the most important things with all of the above points is to **be consistent**! Use the same naming techniques, indentation, documentation style, etc., throughout the whole file.

N Program Testing

To have any confidence that your program is going to work as specified, you will have to test it. A well-tested program cannot guarantee that your program will perform correctly at all times (except in the most trivial cases), but it reduces the likelihood that the program contains a bug.

There are two stages in testing your program: that which you do during the development of your program; and that which you do when you think the program is complete. It is *not* advisable to wait until you think you have a finished program to begin your testing.

1 Developmental Testing

During the development of your program, you should test each function as it is completed, using stubs as necessary. Test each function over a range of inputs, and in particular, choose inputs that will test each branch of a conditional statement (including the default case), when the function has such a statement. If the function contains loops of any type (including recursion), choose inputs that cause the loop to be executed zero, once and many times.

Usually you will discover one or more problems during this stage of testing. Do not just fix the problem and go on! You should go through the full test once again before you go on, because sometimes fixing one problem can introduce new ones. The testing/debugging process is cyclical — you should only go on to the next stage once you have passed *every* test with the current version of your code.

2 Final Testing

Your final testing is used to make sure that the program works correctly as a whole. After the testing you have done during the development of your program, you should be fairly confident that each of your functions works correctly. Should not this mean that your program will work correctly? Well, in theory, yes, but in practice it is impossible to test every combination of inputs to your functions, so you might find that when you put them all together in a complete program, some unexpected results will occur. This is why this final testing is essential.

If you discover a problem, you will have to track down what is causing it (which is often a non-trivial task), and then fix it. When you have fixed it, again completely re-test the function that you had to modify (to make sure you did not introduce a new problem), and then re-test the program.

Once your program has passed all your testing, you can be reasonably sure that it works correctly. However, except in the most trivial cases, it is impossible to be 100% certain that your program will *always* behave correctly, because it is impossible to test every single combination of inputs.

Choice of inputs can be a very difficult task, but one way of choosing data can be to say to yourself "What sort of things might break by program?" Unfortunately, the programmer often finds it very difficult to think of these things for themself (if they can think of these problems for themselves, they should have already included code to handle these cases). This is why in the workplace, there is usually a person or group of people who do no coding themselves, but are solely responsible for testing. For yourself, unless you are working on a project which has testing as part of the assessment criteria, you may find it useful if you get a friend to test your program, and you test theirs.

3 Debugging

As mentioned above, once you have discovered a problem in your program, the task of finding it and fixing it can be very difficult. The way in which you approach debugging can depend heavily upon the language you are coding in, but below are some ideas you may wish to consider.

- Use comments to narrow down where the problem is occurring — if you comment out a large block of code and the problem disappears, then the problem must be somewhere within that block. Gradually uncomment the block, just a small amount at a time, until the problem re-appears. Then you will know to within a few lines of code where the problem is.
- In some languages, using diagnostic write statements can be useful (such as `fprintf(stderr,...` in C). If you are trying to track down the cause of a *segmentation fault* or *bus error*, make sure that you are using *non-buffered I/O*. Make sure that the messages printed out are as informative as possible. Display the values of any variables being used, so that you can check if their values are what you would expect.
- On-line debugging tools are available for some languages. There are two such programs available for UNIX: **gdb**, which can debug programs written in C, C++ or Modula-2; and **dbx**, which can debug programs written in C, Pascal or FORTRAN 77. There is currently no debugging program that works with Hugs. Volume B of the student manual contains information on gdb, or you can use the on-line manual pages for either of these debuggers if you wish to learn more about them.

O Haskell

The Haskell language was conceived during a meeting held at the 1987 Functional Programming and Computer Architecture conference (FPCA 87). At the time of the conference it was believed that the advancement of functional programming was being stifled by the wide variety of languages available. There were more than a dozen lazy, purely functional languages in existence and none had widespread support (except perhaps Miranda[3]). A committee was formed to design the language. The name *Haskell* was chosen in honour of the mathematician *Haskell Curry*, whose research forms part of the theoretical basis upon which many functional languages are implemented. Haskell is widely used within the functional programming community, and there exists a number of implementations. In 1998 the Haskell community agreed upon a standard definition of the language and supporting libraries. One of the aims of standardisation was to encourage the creation of text books and courses devoted to the language. The resulting language definition is called *Haskell 98*.

Haskell is a lazy functional language with polymorphic higher-order functions, algebraic data types and list comprehensions. It has an extensive module system, and supports ad-hoc polymorphism (via classes). Haskell is *purely functional*, even for I/O. Most Haskell implementations come with a number of libraries supporting arrays, complex numbers, infinite precision integers, operating system interaction, concurrency and mutable data structures. There is a popular interpreter (called Hugs) and many compilers. More information about the Haskell language can be found on following the web-page: `http://www.haskell.org`.

Hugs[4] is a freely available interpreter for Haskell, which runs under UNIX, Macintosh, and Microsoft Windows. One of the main features of Hugs is that it provides an interactive programming environment which allows the programmer to edit scripts, and evaluate arbitrary Haskell expressions. Hugs is based significantly on Mark Jones' Gofer interpreter. More information about the Hugs interpreter can be found on the following web-page:
`http://www.haskell.org/hugs`.

The following chapter serves as a reference guide to the Haskell language (specifically Haskell 98). In particular it concentrates on the content of the Haskell Prelude, which is a standard library accessible by all Haskell programs. The chapter does not give complete coverage to the whole Prelude, but instead concentrates on those aspects most useful to Haskell beginners (however it should serve as a valuable resource to experienced Haskell programmers as well). The first part of the chapter deals with Prelude functions, the second part of the chapter deals with Prelude operators, and the third part of the deals with Prelude classes.

[3] Miranda is a trademark of Research Software, Ltd.
[4] Haskell Users' Gofer System

1 Functions from the Haskell Prelude

abs

type: `abs :: Num a => a -> a`

description: returns the absolute value of a number.

definition:
```
abs x
  | x >= 0 = x
  | otherwise = -x
```

usage:
```
Prelude> abs (-3)
3
```

all

type: `all :: (a -> Bool) -> [a] -> Bool`

description: applied to a predicate and a list, returns `True` if all elements of the list satisfy the predicate, and `False` otherwise. Similar to the function `any`.

definition: `all p xs = and (map p xs)`

usage:
```
Prelude> all (<11) [1..10]
True
Prelude> all isDigit "123abc"
False
```

and

type: `and :: [Bool] -> Bool`

description: takes the logical conjunction of a list of boolean values (see also ‘or’).

definition: `and xs = foldr (&&) True xs`

usage:
```
Prelude> and [True, True, False, True]
False
Prelude> and [True, True, True, True]
True
Prelude> and []
True
```

any

type: `any :: (a -> Bool) -> [a] -> Bool`

description: applied to a predicate and a list, returns `True` if any of the elements of the list satisfy the predicate, and `False` otherwise. Similar to the function `all`.

definition: `any p xs = or (map p xs)`

usage:
```
Prelude> any (<11) [1..10]
True
Prelude> any isDigit "123abc"
True
Prelude> any isDigit "alphabetics"
False
```

atan

type: `atan :: Floating a => a -> a`

description: the trigonometric function inverse tan.

definition: defined internally.

usage:
```
Prelude> atan pi
1.26263
```

break

type: `break :: (a -> Bool) -> [a] -> ([a],[a])`

description: given a predicate and a list, breaks the list into two lists (returned as a tuple) at the point where the predicate is first satisfied. If the predicate is never satisfied then the first element of the resulting tuple is the entire list and the second element is the empty list (`[]`).

definition:
```
break p xs
  = span p' xs
    where
    p' x = not (p x)
```

usage:
```
Prelude> break isSpace "hello there fred"
("hello", " there fred")
Prelude> break isDigit "no digits here"
("no digits here","")
```

ceiling

type: `ceiling :: (RealFrac a, Integral b) => a -> b`

description: returns the smallest integer not less than its argument.

usage:
```
Prelude> ceiling 3.8
4
Prelude> ceiling (-3.8)
-3
```

note: the function `floor` has a related use to `ceiling`.

chr

type: `chr :: Int -> Char`

description: applied to an integer in the range 0 – 255, returns the character whose ascii code is that integer. It is the converse of the function `ord`. An error will result if `chr` is applied to an integer outside the correct range.

definition: defined internally.

usage:
```
Prelude> chr 65
'A'
Prelude> (ord (chr 65)) == 65
True
```

concat

type: `concat :: [[a]] -> [a]`

description: applied to a list of lists, joins them together using the ++ operator.

definition: `concat xs = foldr (++) [] xs`

usage:
```
Prelude> concat [[1,2,3], [4], [], [5,6,7,8]]
[1, 2, 3, 4, 5, 6, 7, 8]
```

cos

type: `cos :: Floating a => a -> a`

description: the trigonometric cosine function, arguments are interpreted to be in radians.

definition: defined internally.

usage:
```
Prelude> cos pi
-1.0
Prelude> cos (pi/2)
-4.37114e-08
```

digitToInt

type: `digitToInt :: Char -> Int`

description: converts a digit character into the corresponding integer value of the digit.

definition:
```
digitToInt :: Char -> Int
digitToInt c
  | isDigit c            =  fromEnum c - fromEnum '0'
  | c >= 'a' && c <= 'f' =  fromEnum c - fromEnum 'a' + 10
  | c >= 'A' && c <= 'F' =  fromEnum c - fromEnum 'A' + 10
  | otherwise            =  error "Char.digitToInt: not a digi
```

usage:
```
Prelude> digitToInt '3'
3
```

div

type: `div :: Integral a => a -> a -> a`

description: computes the integer division of its integral arguments.

definition: defined internally.

usage:
```
Prelude> 16 ‘div‘ 9
1
```

doReadFile

type: `doReadFile :: String -> String`

description: given a filename as a string, returns the contents of the file as a string. Returns an error if the file cannot be opened or found.

definition: defined internally.

usage:
```
Prelude> doReadFile "foo.txt"
"This is a small text file,\ncalled foo.txt.\n"
```

note: This is *not* a standard Haskell function. You must `import` the `MULib.hs` module to use this function.

drop

type: `drop :: Int -> [a] -> [a]`

description: applied to a number and a list, returns the list with the specified number of elements removed from the front of the list. If the list has less than the required number of elements then it returns `[]`.

definition:
```
drop 0 xs           = xs
drop _ []           = []
drop n (_:xs) | n>0 = drop (n-1) xs
drop _ _            = error "PreludeList.drop: negative argument"
```

usage:
```
Prelude> drop 3 [1..10]
[4, 5, 6, 7, 8, 9, 10]
Prelude> drop 4 "abc"
""
```

dropWhile

type: `dropWhile :: (a -> Bool) -> [a] -> [a]`

description: applied to a predicate and a list, removes elements from the front of the list while the predicate is satisfied.

definition:
```
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = (x:xs)
```

usage:
```
Prelude> dropWhile (<5) [1..10]
[5, 6, 7, 8, 9, 10]
```

elem

type: `elem :: Eq a => a -> [a] -> Bool`

description: applied to a value and a list returns `True` if the value is in the list and `False` otherwise. The elements of the list must be of the same type as the value.

definition: `elem x xs = any (== x) xs`

usage:
```
Prelude> elem 5 [1..10]
True
Prelude> elem "rat" ["fat", "cat", "sat", "flat"]
False
```

error

type: `error :: String -> a`

description: applied to a string creates an error value with an associated message. Error values are equivalent to the undefined value (`undefined`), any attempt to access the value causes the program to terminate and print the string as a diagnostic.

definition: defined internally.

usage:
```
error "this is an error message"
```

exp

type: `exp :: Floating a => a -> a`

description: the exponential function (`exp n` is equivalent to e^n).

definition: defined internally.

usage:
```
Prelude> exp 1
2.71828
```

filter

type: `filter :: (a -> Bool) -> [a] -> [a]`

description: applied to a predicate and a list, returns a list containing all the elements from the argument list that satisfy the predicate.

definition:
```
filter p xs = [k | k <- xs, p k]
```

usage:
```
Prelude> filter isDigit "fat123cat456"
"123456"
```

flip

type:
```
flip ::  (a -> b -> c) -> b -> a -> c
```

description: applied to a binary function, returns the same function with the order of the arguments reversed.

definition:
```
flip f x y = f y x
```

usage:
```
Prelude> flip elem [1..10] 5
True
```

floor

type:
```
floor ::  (RealFrac a, Integral b) => a -> b
```

description: returns the largest integer not greater than its argument.

usage:
```
Prelude> floor 3.8
3
Prelude> floor (-3.8)
-4
```

note: the function `ceiling` has a related use to `floor`.

foldl

type:
```
foldl ::  (a -> b -> a) -> a -> [b] -> a
```

description: folds up a list, using a given binary operator and a given start value, in a left associative manner.

`foldl op r [a, b, c]` → ((r ‘op‘ a) ‘op‘ b) ‘op‘ c

definition:
```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

usage:
```
Prelude> foldl (+) 0 [1..10]
55
Prelude> foldl (flip (:)) [] [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

foldl1

type: `foldl1 :: (a -> a -> a) -> [a] -> a`

description: folds left over non–empty lists.

definition:
```
foldl1 f (x:xs) = foldl f x xs
```

usage:
```
Prelude> foldl1 max [1, 10, 5, 2, -1]
10
```

foldr

type: `foldr :: (a -> b -> b) -> b -> [a] -> b`

description: folds up a list, using a given binary operator and a given start value, in a right associative manner.

`foldr op r [a, b, c]` → a ‘op‘ (b ‘op‘ (c ‘op‘ r))

definition:
```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

usage:
```
Prelude> foldr (++) [] ["con", "cat", "en", "ate"]
"concatenate"
```

foldr1

type: `foldr1 :: (a -> a -> a) -> [a] -> a`

description: folds right over non–empty lists.

definition:
```
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

usage:
```
Prelude> foldr1 (*) [1..10]
3628800
```

fromInt

type: `fromInt :: Num a => Int -> a`

description: Converts from an `Int` to a numeric type which is in the class `Num`.

usage:
```
Prelude> (fromInt 3)::Float
3.0
```

fromInteger

type: `fromInteger :: Num a => Integer -> a`

description: Converts from an `Integer` to a numeric type which is in the class `Num`.

usage:
```
Prelude> (fromInteger 10000000000)::Float
1.0e+10
```

fst

type: `fst :: (a, b) -> a`

description: returns the first element of a two element tuple.

definition: `fst (x, _) = x`

usage:
```
Prelude> fst ("harry", 3)
"harry"
```

head

type: `head :: [a] -> a`

description: returns the first element of a non–empty list. If applied to an empty list an error results.

definition: `head (x:_) = x`

usage:
```
Prelude> head [1..10]
1
Prelude> head ["this", "and", "that"]
"this"
```

id

type: `id :: a -> a`

description: the identity function, returns the value of its argument.

definition: `id x = x`

usage:
```
Prelude> id 12
12
Prelude> id (id "fred")
"fred"
Prelude> (map id [1..10]) == [1..10]
True
```

init

type: `init :: [a] -> [a]`

description: returns all but the last element of its argument list. The argument list must have at least one element. If `init` is applied to an empty list an error occurs.

definition:
```
init [x] = []
init (x:xs) = x : init xs
```

usage:
```
Prelude> init [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

isAlpha

type: `isAlpha :: Char -> Bool`

description: applied to a character argument, returns `True` if the character is alphabetic, and `False` otherwise.

definition:
```
isAlpha c = isUpper c || isLower c
```

usage:
```
Prelude> isAlpha 'a'
True
Prelude> isAlpha '1'
False
```

isDigit

type: `isDigit :: Char -> Bool`

description: applied to a character argument, returns `True` if the character is a numeral, and `False` otherwise.

definition:
```
isDigit c = c >= '0' && c <= '9'
```

usage:
```
Prelude> isDigit '1'
True
Prelude> isDigit 'a'
False
```

isLower

type: `isLower :: Char -> Bool`

description: applied to a character argument, returns `True` if the character is a lower case alphabetic, and `False` otherwise.

definition:
```
isLower c = c >= 'a' && c <= 'z'
```

usage:
```
Prelude> isLower 'a'
True
Prelude> isLower 'A'
False
Prelude> isLower '1'
False
```

isSpace

type: `isSpace :: Char -> Bool`

description: returns `True` if its character argument is a whitespace character and `False` otherwise.

definition:
```
isSpace c  = c == ' '  || c == '\t' || c == '\n' ||
             c == '\r' || c == '\f' || c == '\v'
```

usage:
```
Prelude> dropWhile isSpace "   \nhello  \n"
"hello  \n"
```

isUpper

type: `isUpper :: Char -> Bool`

description: applied to a character argument, returns `True` if the character is an upper case alphabetic, and `False` otherwise.

definition:
```
isDigit c = c >= 'A' && c <= 'Z'
```

usage:
```
Prelude> isUpper 'A'
True
Prelude> isUpper 'a'
False
Prelude> isUpper '1'
False
```

iterate

type: `iterate :: (a -> a) -> a -> [a]`

description: `iterate f x` returns the infinite list `[x, f(x), f(f(x)), ...]`.

definition:
```
iterate f x = x : iterate f (f x)
```

usage:
```
Prelude> iterate (+1) 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, .....
```

last

type: `last :: [a] -> a`

description: applied to a non–empty list, returns the last element of the list.

definition:
```
last [x] = x
last (_:xs) = last xs
```

usage:
```
Prelude> last [1..10]
10
```

length

type:
```
length ::  [a] -> Int
```

description: returns the number of elements in a finite list.

definition:
```
length [] = 0
length (x:xs) = 1 + length xs
```

usage:
```
Prelude> length [1..10]
10
```

lines

type:
```
lines ::  String -> [String]
```

description: applied to a list of characters containing newlines, returns a list of lists by breaking the original list into lines using the newline character as a delimiter. The newline characters are removed from the result.

definition:
```
lines [] = []
lines (x:xs)
  = l : ls
  where
  (l, xs') = break (== '\n') (x:xs)
  ls
    | xs' == [] = []
    | otherwise = lines (tail xs')
```

usage:
```
Prelude> lines "hello world\nit's me,\neric\n"
["hello world", "it's me,", "eric"]
```

log

type:
```
log ::  Floating a => a -> a
```

description: returns the natural logarithm of its argument.

definition: defined internally.

usage:
```
Prelude> log 1
0.0
Prelude> log 3.2
1.16315
```

map

type: `map :: (a -> b) -> [a] -> [b]`

description: given a function, and a list of any type, returns a list where each element is the result of applying the function to the corresponding element in the input list.

definition:
```
map f xs = [f x | x <- xs]
```

usage:
```
Prelude> map sqrt [1..5]
[1.0, 1.41421, 1.73205, 2.0, 2.23607]
```

max

type: `max :: Ord a => a -> a -> a`

description: applied to two values of the same type which have an ordering defined upon them, returns the maximum of the two elements according to the operator >=.

definition:
```
max x y
  | x >= y = x
  | otherwise = y
```

usage:
```
Prelude> max 1 2
2
```

maximum

type: `maximum :: Ord a => [a] -> a`

description: applied to a non–empty list whose elements have an ordering defined upon them, returns the maximum element of the list.

definition:
```
maximum xs = foldl1 max xs
```

usage:
```
Prelude> maximum [-10, 0 , 5, 22, 13]
22
```

min

type: `min :: Ord a => a -> a -> a`

description: applied to two values of the same type which have an ordering defined upon them, returns the minimum of the two elements according to the operator <=.

definition:
```
min x y
  | x <= y = x
  | otherwise = y
```

usage:
```
Prelude> min 1 2
1
```

minimum

type: `minimum :: Ord a => [a] -> a`

description: applied to a non–empty list whose elements have an ordering defined upon them, returns the minimum element of the list.

definition:
```
minimum xs = foldl1 min xs
```

usage:
```
Prelude> minimum [-10, 0 , 5, 22, 13]
-10
```

mod

type: `mod :: Integral a => a -> a -> a`

description: returns the modulus of its two arguments.

definition: defined internally.

usage:
```
Prelude> 16 ‘mod‘ 9
7
```

not

type: `not :: Bool -> Bool`

description: returns the logical negation of its boolean argument.

definition:
```
not True = False
not False = True
```

usage:
```
Prelude> not (3 == 4)
True
Prelude> not (10 > 2)
False
```

or

type: `or :: [Bool] -> Bool`

description: applied to a list of boolean values, returns their logical disjunction (see also ‘and’).

definition:
```
or xs = foldr (||) False xs
```

usage:
```
Prelude> or [False, False, True, False]
True
Prelude> or [False, False, False, False]
False
Prelude> or []
False
```

ord

type: `ord :: Char -> Int`

description: applied to a character, returns its ascii code as an integer.

definition: defined internally.

usage:
```
Prelude> ord 'A'
65
Prelude> (chr (ord 'A')) == 'A'
True
```

pi

type: `pi :: Floating a => a`

description: the ratio of the circumference of a circle to its diameter.

definition: defined internally.

usage:
```
Prelude> pi
3.14159
Prelude> cos pi
-1.0
```

putStr

type: `putStr :: String -> IO ()`

description: takes a string as an argument and returns an I/O action as a result. A side-effect of applying `putStr` is that it causes its argument string to be printed to the screen.

definition: defined internally.

usage:
```
Prelude> putStr "Hello World\nI'm here!"
Hello World
I'm here!
```

product

type: `product :: Num a => [a] -> a`

description: applied to a list of numbers, returns their product.

definition: `product xs = foldl (*) 1 xs`

usage:
```
Prelude> product [1..10]
3628800
```

repeat

type: `repeat :: a -> [a]`

description: given a value, returns an infinite list of elements the same as the value.

definition:
```
repeat x
  = xs
  where xs = x:xs
```

usage:
```
Prelude> repeat 12
[12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12 ....
```

replicate

type: `replicate :: Int -> a -> [a]`

description: given an integer (positive or zero) and a value, returns a list containing the specified number of instances of that value.

definition:
```
replicate n x = take n (repeat x)
```

usage:
```
Prelude> replicate 3 "apples"
["apples", "apples", "apples"]
```

reverse

type: `reverse :: [a] -> [a]`

description: applied to a finite list of any type, returns a list of the same elements in reverse order.

definition:
```
reverse = foldl (flip (:)) []
```

usage:
```
Prelude> reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

round

type: `round :: (RealFrac a, Integral b) => a -> b`

description: rounds its argument to the nearest integer.

usage:
```
Prelude> round 3.2
3
Prelude> round 3.5
4
Prelude> round (-3.2)
-3
```

show

type: `show :: Show a => a -> String`

description: converts a value (which must be a member of the `Show` class), to its string representation.

definition: defined internally.

usage:
```
Prelude> "six plus two equals " ++ (show (6 + 2))
"six plus two equals 8"
```

sin

type: `sin :: Floating a => a -> a`

description: the trigonometric sine function, arguments are interpreted to be in radians.

definition: defined internally.

usage:
```
Prelude> sin (pi/2)
1.0
Prelude> ((sin pi)^2) + ((cos pi)^2)
1.0
```

snd

type: `snd :: (a, b) -> b`

description: returns the second element of a two element tuple.

definition: `snd (_, y) = y`

usage:
```
Prelude> snd ("harry", 3)
3
```

sort

type: `sort :: Ord a => [a] -> [a]`

description: sorts its argument list in ascending order. The items in the list must be in the class `Ord`.

usage:
```
List> sort [1, 4, -2, 8, 11, 0]
[-2,0,1,4,8,11]
```

note: This is *not* defined within the Prelude. You must `import` the `List.hs` module to use this function.

span

type:
```
span ::  (a -> Bool) -> [a] -> ([a],[a])
```

description: given a predicate and a list, splits the list into two lists (returned as a tuple) such that elements in the first list are taken from the head of the list while the predicate is satisfied, and elements in the second list are the remaining elements from the list once the predicate is not satisfied.

definition:
```
span p [] = ([],[])
span p xs@(x:xs')
  | p x = (x:ys, zs)
  | otherwise = ([],xs)
    where (ys,zs) = span p xs'
```

usage:
```
Prelude> span isDigit "123abc456"
("123", "abc456")
```

splitAt

type:
```
splitAt ::  Int -> [a] -> ([a],[a])
```

description: given an integer (positive or zero) and a list, splits the list into two lists (returned as a tuple) at the position corresponding to the given integer. If the integer is greater than the length of the list, it returns a tuple containing the entire list as its first element and the empty list as its second element.

definition:
```
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs)
  | n > 0 = (x:xs',xs'')
    where
    (xs',xs'') = splitAt (n-1) xs
splitAt _ _ = error "PreludeList.splitAt: negative argument"
```

usage:
```
Prelude> splitAt 3 [1..10]
([1, 2, 3], [4, 5, 6, 7, 8, 9, 10])
Prelude> splitAt 5 "abc"
("abc", "")
```

sqrt

type:
```
sqrt ::  Floating a => a -> a
```

description: returns the square root of a number.

definition:
```
sqrt x = x ** 0.5
```

usage:
```
Prelude> sqrt 16
4.0
```

subtract

type: `subtract :: Num a => a -> a -> a`

description: subtracts its first argument from its second argument.

definition:
```
subtract = flip (-)
```

usage:
```
Prelude> subtract 7 10
3
```

sum

type: `sum :: Num a => [a] -> a`

description: computes the sum of a finite list of numbers.

definition:
```
sum xs = foldl (+) 0 xs
```

usage:
```
Prelude> sum [1..10]
55
```

tail

type: `tail :: [a] -> [a]`

description: applied to a non–empty list, returns the list without its first element.

definition:
```
tail (_:xs) = xs
```

usage:
```
Prelude> tail [1,2,3]
[2,3]
Prelude> tail "hugs"
"ugs"
```

take

type: `take :: Int -> [a] -> [a]`

description: applied to an integer (positive or zero) and a list, returns the specified number of elements from the front of the list. If the list has less than the required number of elements, `take` returns the entire list.

definition:
```
take 0 _ = []
take _ []= []
take n (x:xs)
  | n > 0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"
```

usage:
```
Prelude> take 4 "goodbye"
"good"
Prelude> take 10 [1,2,3]
[1,2,3]
```

takeWhile

type: `takewhile :: (a -> Bool) -> [a] -> [a]`

description: applied to a predicate and a list, returns a list containing elements from the front of the list while the predicate is satisfied.

definition:
```
takeWhile p [] = []
takeWhile p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []
```

usage:
```
Prelude> takeWhile (<5) [1, 2, 3, 10, 4, 2]
[1, 2, 3]
```

tan

type: `tan :: Floating a => a -> a`

description: the trigonometric function tan, arguments are interpreted to be in radians.

definition: defined internally.

usage:
```
Prelude> tan (pi/4)
1.0
```

toLower

type: `toLower :: Char -> Char`

description: converts an uppercase alphabetic character to a lowercase alphabetic character. If this function is applied to an argument which is not uppercase the result will be the same as the argument unchanged.

definition:
```
toLower c
  | isUpper c = toEnum (fromEnum c - fromEnum 'A' + fromEnum '
  | otherwise = c
```

usage:
```
Prelude> toLower 'A'
'a'
Prelude> toLower '3'
'3'
```

toUpper

type: `toUpper :: Char -> Char`

description: converts a lowercase alphabetic character to an uppercase alphabetic character. If this function is applied to an argument which is not lowercase the result will be the same as the argument unchanged.

definition:
```
toUpper c
  | isLower c = toEnum (fromEnum c - fromEnum 'a' + fromEnum 'A')
  | otherwise = c
```

usage:
```
Prelude> toUpper 'a'
'A'
Prelude> toUpper '3'
'3'
```

truncate

type: `truncate :: (RealFrac a, Integral b) => a -> b`

description: drops the fractional part of a floating point number, returning only the integral part.

usage:
```
Prelude> truncate 3.2
3
Prelude> truncate (-3.2)
-3
```

note:

unlines

type: `unlines :: [String] -> String`

description: converts a list of strings into a single string, placing a newline character between each of them. It is the converse of the function `lines`.

definition:
```
unlines xs
  = concat (map addNewLine xs)
  where
  addNewLine l = l ++ "\n"
```

usage:
```
Prelude> unlines ["hello world", "it's me,", "eric"]
"hello world\nit's me,\neric\n"
```

until

type: `until :: (a -> Bool) -> (a -> a) -> a -> a`

description: given a predicate, a unary function and a value, it recursively re–applies the function to the value until the predicate is satisfied. If the predicate is never satisfied `until` will not terminate.

definition:
```
until p f x
  | p x = x
  | otheriwise = until p f (f x)
```

usage:
```
Prelude> until (>1000) (*2) 1
1024
```

unwords

type: `unwords :: [String] -> String`

description: concatenates a list of strings into a single string, placing a single space between each of them.

definition:
```
unwords [] = []
unwords ws
  = foldr1 addSpace ws
  where
  addSpace w s = w ++ (' ':s)
```

usage:
```
Prelude> unwords ["the", "quick", "brown", "fox"]
"the quick brown fox"
```

words

type: `words :: String -> [String]`

description: breaks its argument string into a list of words such that each word is delimited by one or more whitespace characters.

definition:
```
words s
  | findSpace == [] = []
  | otherwise = w : words s''
  where
  (w, s'') = break isSpace findSpace
  findSpace = dropWhile isSpace s
```

usage:
```
Prelude> words "the quick brown\n\nfox"
["the", "quick", "brown", "fox"]
```

zip

type: `zip :: [a] -> [b] -> [(a,b)]`

description: applied to two lists, returns a list of pairs which are formed by tupling together corresponding elements of the given lists. If the two lists are of different length, the length of the resulting list is that of the shortest.

definition:
```
zip xs ys
  = zipWith pair xs ys
  where
  pair x y = (x, y)
```

usage:
```
Prelude> zip [1..6] "abcd"
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

zipWith

type: `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

description: applied to a binary function and two lists, returns a list containing elements formed be applying the function to corresponding elements in the lists.

definition:
```
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []
```

usage:
```
Prelude> zipWith (+) [1..5] [6..10]
[7, 9, 11, 13, 15]
```

2 A Description of Standard Haskell Operators

Operators are simply functions of one or two arguments. Operators usually get written between their arguments (called infix notation), rather than to the left of them. Many operators have symbolic names (like + for plus), however this is out of convention rather than necessity. Others have completely textual names (such as `` `div` `` for integer division).

The following table lists many useful operators defined in the Prelude. Definitions of *associativity* and *binding power* are given after the table.

symbol	*behaviour*	*type*	*assoc–iativity*	*binding power*
`!!`	list subscript	`[a] -> Int -> a`	left	9
`.`	function compose	`(a -> b) -> (c -> a) -> c -> b`	right	9
`^`	exponentiation	`(Integral b, Num a) => a -> b -> a`	right	8
`**`	exponentiation	`Floating a => a -> a -> a`	right	8
`%`	ratio	`Integral a => a -> a -> Ratio a`	left	7
`*`	multiplication	`Num a => a -> a -> a`	left	7
`/`	division	`Fractional a => a -> a -> a`	left	7
`` `div` ``	integer division	`Integral a => a -> a -> a`	left	7
`` `mod` ``	modulus	`Integral a => a -> a -> a`	left	7
`+`	plus	`Num a => a -> a -> a`	left	6
`-`	minus	`Num a => a -> a -> a`	left	6
`:`	list construct	`a -> [a] -> [a]`	right	5
`++`	concatenate	`MonadPlus a => a b -> a b -> a b` (for lists: `[a] -> [a] -> [a]`)	right	5
`/=`	not equal	`Eq a => a -> a -> Bool`	non	4
`==`	equal	`Eq a => a -> a -> Bool`	non	4
`<`	less than	`Ord a => a -> a -> Bool`	non	4
`<=`	less than or equal	`Ord a => a -> a -> Bool`	non	4
`>`	greater than	`Ord a => a -> a -> Bool`	non	4
`>=`	greater than or equal	`Ord a => a -> a -> Bool`	non	4
`` `elem` ``	list contains	`Eq a => a -> [a] -> Bool`	non	4
`` `notElem` ``	list not contains	`Eq a => a -> [a] -> Bool`	non	4
`&&`	logical and	`Bool -> Bool -> Bool`	right	3
`||`	logical or	`Bool -> Bool -> Bool`	right	3

The higher the binding power the more tightly the operator binds to its arguments.
Function application has a binding power of 10,
and so takes preference over any other operator application.

Associativity: sequences of operator applications are allowed in Haskell for the convenience of the programmer. However, in some circumstances the meaning of such a sequence can be ambiguous. For example, we could interpret the expression `8 - 2 - 1` in two ways, either as `(8 - 2) - 1`, or as `8 - (2 - 1)` (each interpretation having a different value). Associativity tells us whether a sequence of a particular operator should be bracketed to the left or to the right. As it happens, the minus operator (`-`) is left associative, and so Haskell chooses the first of the alternative interpretations as the meaning of the above expression. The choice of associativity for an operator is quite arbitrary, however, they usually follow conventional mathematical notation. Note that some operators are *non-associative*, which means that they cannot be applied in sequence. For example, the equality operator (`==`) is non–associative, and therefore the following expression is not allowed in Haskell: `2 == (1 + 1) == (3 - 1)`.

Binding Power: Haskell expressions may also contain a mixture of operator applications which can lead to ambiguities that the rules of associativity cannot solve. For example, we could interpret the expression `3 - 4 * 2` in two ways, either as `(3 - 4) * 2`, or as `3 - (4 * 2)` (each interpretation having a different value). Binding power tells us which operators take precedence in an expression containing a mixture of operators. The multiplication operator (`*`), has a binding power of 7 (out of a possible 10), and the minus operator (`-`) has a binding power of 6. Therefore the multiplication operator takes precedence over the minus operator, and thus Haskell chooses the second of the alternative interpretations as the meaning of the above expression. All operators must have a binding power assigned to them which ranges from 1 to 10. Function application takes prededence over everything else in an expression, and so the expression `reverse [1..10] ++ [0]` is interpreted as `(reverse [1..10]) ++ [0]`, rather than `reverse ([1..10] ++ [0])`.

3 Using the Standard Haskell Operators

`!!`

description: given a list and a number, returns the element of the list whose position is the same as the number.

usage:
```
Prelude> [1..10] !! 0
1
Prelude> "a string" !! 3
't'
```

notes: the valid subscripts for a list l are: $0 \leq subscript \leq ((length\ l) - 1)$. Therefore, negative subscripts are not allowed, nor are subsripts greater than one less than the length of the list argument. Subscripts out of this range will result in a program error.

`.`

description: composes two functions into a single function.

usage:
```
Prelude> (sqrt . sum ) [1,2,3,4,5]
3.87298
```

notes: `(f.g.h) x` is equivalent to `f (g (h x))`.

`**`

description: raises its first argument to the power of its second argument. The arguments must be in the `Floating` numerical type class, and the result will also be in that class.

usage:
```
Prelude> 3.2**pi
38.6345
```

`^`

description: raises its first argument to the power of its second argument. The first argument must be a member of the `Num` typeclass, and the second argument must be a member of the `Integral` typeclass. The result will be of the same type as the first argument.

usage:
```
Prelude> 3.2^4
104.858
```

`%`

description: takes two numbers in the `Integral` typeclass and returns the most simple ratio of the two.

usage:
```
Prelude> 20 % 4
5 % 1
Prelude> (5 % 4)^2
25 % 16
```

`*`

description: returns the multiple of its two arguments.

usage:
```
Prelude> 6 * 2.0
12.0
```

`/`

description: returns the result of dividing its first argument by its second. Both arguments must in the type class `Fractional`.

usage:
```
Prelude> 12.0 / 2
6.0
```

`‘div‘`

description: returns the integral division of the first argument by the second argument. Both arguments must be in the type class `Integral`.

usage:
```
Prelude> 10 ‘div‘ 3
3
Prelude> 3 ‘div‘ 10
0
```

notes: `‘div‘` is integer division such that the result is truncated towards negative infinity.

```
Prelude> (-12) ‘div‘ 5
-3
Prelude> 12 ‘div‘ 5
2
```

`‘mod‘`

description: returns the integral remainder after dividing the first argument by the second. Both arguments must be in the type class `Integral`.

usage:
```
Prelude> 10 ‘mod‘ 3
1
Prelude> 3 ‘mod‘ 10
3
```

`+`

description: returns the addition of its arguments.

usage:
```
Prelude> 3 + 4
7
Prelude> (4 % 5) + (1 % 5)
1 % 1
```

`-`

description: returns the substraction of its second argument from its first.

usage:
```
Prelude> 4 - 3
1
Prelude> 4 - (-3)
7
```

`:`

description: prefixes an element onto the front of a list.

usage:
```
Prelude> 1:[2,3]
[1,2,3]
Prelude> True:[]
[True]
Prelude> ’h’:"askell"
"haskell"
```

`++`

description: appends its second list argument onto the end of its first list argument.

usage:
```
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
Prelude> "foo " ++ "was" ++ " here"
"foo was here"
```

`/=`

description: is `True` if its first argument is not equal to its second argument, and `False` otherwise. Equality is defined by the `==` operator. Both of its arguments must be in the `Eq` type class.

usage:
```
Prelude> 3 /= 4
True
Prelude> [1,2,3] /= [1,2,3]
False
```

`==`

description: is `True` if its first argument is equal to its second argument, and `False` otherwise. Equality is defined by the `==` operator. Both of its arguments must be in the `Eq`

usage:
```
Prelude> 3 == 4
False
Prelude> [1,2,3] == [1,2,3]
True
```

`<`

description: returns `True` if its first argument is strictly less than its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

usage:
```
Prelude> 1 < 2
True
Prelude> 'a' < 'z'
True
Prelude> True < False
False
```

`<=`

description: returns `True` if its first argument is less than or equal to its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

usage:
```
Prelude> 3 <= 4
True
Prelude> 4 <= 4
True
Prelude> 5 <= 4
False
```

`>`

description:

usage: returns `True` if its first argument is strictly greater than its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

```
Prelude> 2 > 1
True
Prelude> 'a' > 'z'
False
Prelude> True > False
True
```

`>=`

description:

usage: returns `True` if its first argument is greater than or equal to its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

```
Prelude> 4 >= 3
True
Prelude> 4 >= 4
True
Prelude> 4 >= 5
False
```

`‘elem‘`

description: returns `True` if its first argument is an element of the list as its second argument, and `False` otherwise.

usage:

```
Prelude> 3 ‘elem‘ [1,2,3]
True
Prelude> 4 ‘elem‘ [1,2,3]
False
```

`‘notElem‘`

description: returns `True` if its first argument is *not* an element of the list as its second argument.

usage:
```
Prelude> 3 ‘notElem‘ [1,2,3]
False
Prelude> 4 ‘notElem‘ [1,2,3]
True
```

`&&`

description: returns the logical conjunction of its two boolean arguments.

usage:
```
Prelude> True && True
True
Prelude> (3 < 4) && (4 < 5) && False
False
```

`||`

description: returns the logical disjunction of its two boolean arguments.

usage:
```
Prelude> True || False
True
Prelude> (3 < 4) || (4 > 5) || False
True
```

4 Type Classes from the Haskell Prelude

Eq

description: Types which are instances of this class have equality defined upon them. This means that all elements of such types can be compared for equality.

instances:
- All Prelude types except `IO` and functions.

notes: Functions which use the equality operators (`==`, `/=`) or the functions `elem` or `notElem` will often be subject to the `Eq` type class, thus requiring the constraint `Eq a =>` in the type signature for that function.

Ord

description: Types which are instances of this class have a complete ordering defined upon them.

instances:
- All Prelude types except `IO`, functions, and `IOError`.

notes: Functions which use the comparison operators (`>`, `<`, `>=`, `<=`), or the functions `max`, `min`, `maximum` or `minimum` will often be subject to the `Ord` type class, thus requiring the constraint `Ord a =>` in the type signature for that function.

Enum

description: Types which are instances of this class can be enumerated. This means that all elements of such types have a mapping to a unique integer, thus the elements of the type must be sequentially ordered.

instances:
- `Bool`
- `Char`
- `Int`
- `Integer`
- `Float`
- `Double`

notes: Functions which use dot-dot notation (eg `[1,3 .. y]`) in list comprehensions will often be subject to the `Enum` type class, thus requiring the constraint `Enum a =>` in the type signature for that function.

Show

description: Types which are instances of this class have a printable representation. This means that all elements of such types can be given as arguments to the function `show`.

instances:
- All Prelude types.

notes: Functions which use the function `show` will often be subject to the `Show` type class, thus requiring the constraint `Show a =>` in the type signature for that function.

Read

description: Types which are instances of this class allow a string representation of all elements of the type to be converted to the corresponding element.

instances:
- All Prelude types except `IO` and functions.

notes: Functions which use the function `read` will often be subject to the `Read` type class, thus requiring the constraint `Read a =>` in the type signature for that function.

Num

description: This is the parent class for all the numeric classes. Any type which is an instance of this class must have basic numeric operators (such as plus, minus and multiply) defined on them, and must be able to be converted from an `Int` or `Integer` to an element of the type.

instances:
- `Int`
- `Integer`
- `Float`
- `Double`

notes: Functions which perform operations which are applicable to all numeric types, but not to other non–numeric types will often be subject to the `Num` type class, thus requiring the constraint `Num a =>` in the type signature for that function.

Real

description: This class covers all the numeric types whose elements can be expressed as a ratio.

instances:
- `Int`
- `Integer`
- `Float`
- `Double`

Fractional

description: This class covers all the numeric types whose elements are fractional. All such types must have division defined upon them, they must have a reciprocal, and must be convertible from rational numbers, and double precision floating point numbers.

instances:
- `Float`

- `Double`

notes: Functions which use the division operator (`/`) will often be subject to the `Fractional` type class, thus requiring the constraint `Fractional a =>` in the type signature for that function.

Integral

description: This class covers all the numeric types whose elements are integral.

instances:
- `Int`
- `Integer`

notes: Functions which use the operators `div` or `mod` will often be subject to the `Integral` type class, thus requiring the constraint `Integral a =>` in the type signature for that function.

Floating

description: This class covers all the numeric types whose elements are floating point numbers.

instances:
- `Float`
- `Double`

notes: Functions which use the constant `pi` or the functions `exp`, `log`, `sqrt`, `sin`, `cos` or `tan` will often be subject to the `Floating` type class, thus requiring the constraint `Floating a =>` in the type signature for that function.

5 The Haskell Prelude Class Hierarchy

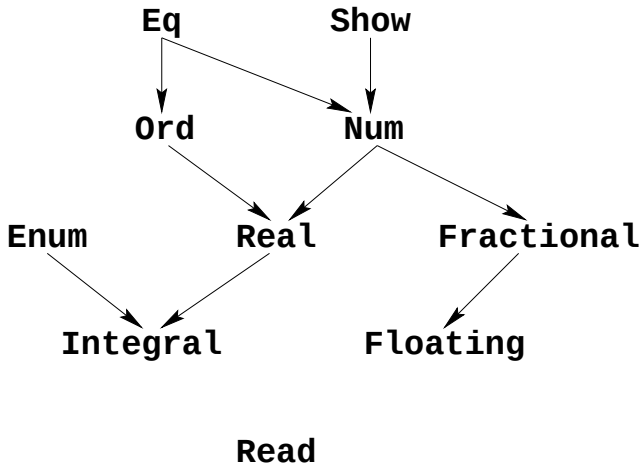


Figure 1: A sample of the class hierarchy from the Haskell Prelude

Figure 1 illustrates a sample of the type class hierarchy from the Haskell Prelude. Arrows in the diagram represent the ordering of classes in the hierarchy. For example, for a type to be in the class `Ord` it must also be in the class `Eq`. Note that the class `Read` is separate from the rest of the hierarchy.

P Commonly Used UNIX Commands

Note: Where no example is provided, the command is to be run with no command-line arguments.

1 Access

login Sign onto the system

logout Sign off the system

^D (Control-D) Exit current shell

chpw Change current password

rlogin Sign onto a remote system
e.g. **rlogin toaster**
Begins sign-on process to machine **toaster**.

2 Job Control

^C (Control-C) Terminate current foreground job

^Z (Control-Z) Suspend current foreground job

fg Bring suspended job into foreground
e.g. **fg %n**
Brings suspended job with job ID number *n* into the foreground. Default (**fg** by itself) is to bring job number 1 into the foreground.

jobs List current jobs and their status

ps List processes running on behalf of user

kill Terminate specified job
e.g. **kill %1**
Terminates process with job ID number **1** found by using **jobs**.

3 Environment

term Shell variable determining screen output mode
e.g. **set term=vt100**
Sets terminal type to a common setting.

stty Set the options for a terminal
e.g. **stty erase ^?**
Sets the **^?** character to be backspace.

4 Entering Commands

! Start a history substitution
e.g. **!62**
Re-runs command number 62 from history (see **history** below).
e.g. **!-3**
Re-runs the command three before the current one in the history.

!! Refer to the previous command

^^ Substitute on previous command
e.g. If the previous command was **ls /home/stude** then **^ls^cd** will run the command **cd /home/stude**.

history Display user's previous commands list with line numbers

5 Redirection & Piping

> Redirect output, creating file if necessary.
e.g. **ls > dir_list**
Stores the output of the **ls** command in the file *dir_list*.

>> Redirect and append output to file, creating file if necessary
e.g. **ls >> dir_list**
Appends the output of the **ls** command onto the end of the file *dir_list*.

< Redirect input from file
e.g. **mail foo < letter**
Sends user *foo* the text file *letter* via electronic mail.

2>&1 Combine Standard Output and Standard Error
e.g. **ls > dir_list 2>&1**
Directs both standard output and standard error to the file *dir_list*.

| Channel output of one command into input of another
e.g. **ls | more**
Sends output of **ls** through the program **more**.

6 Editing

vi Visual editor
e.g. **vi my_proj.c**
Opens the file *my_proj.c* for editing, creating it if necessary.

emacs Real-time display editor
e.g. **emacs my_proj.c**
Opens the file *my_proj.c* for editing, creating it if necessary.

pico Simple text editor
e.g. **pico my_proj.c**
Opens the file *my_proj.c* for editing, creating it if necessary.

7 Displaying Text Files

more Display text file contents one screen at a time.
e.g. **more my_file**
Displays contents of the file *my_file*, one screen at a time.

less Display text file contents one screen at a time. Allows forwards and backwards scrolling and searches.
e.g. **less my_file**
Displays contents of the file *my_file*, one screen at a time.

cat Display contents of a text file
e.g. **cat my_file**
Displays contents of the file *my_file*.

^S Pause screen output

^Q Resume paused screen output. Used in conjunction with **^S**.

8 Printing

lpr Send files to the printer
e.g. **lpr my_file**
Sends the file *my_file* to be printed.

lpq Show the printer queue
e.g. **lpq**
Displays printer queue information.

lprm Remove requests from the printer queue
e.g. **lprm 1024**
Removes printer job number *1024* (see output of **lpq**) from printer queue.

9 File Utilities

cp Copy file(s) from one location to another
e.g. **cp file_1 ../file_2**
Creates a new file in the parent directory called *file_2* into which the contents of *file_1* are placed. *file_1* is unchanged.

mv Move file(s) around the file system
e.g **mv file_1 ../file_2**
Moves contents of *file_1* to a file in the parent directory called *file_2*. *file_1* is removed.
e.g. **mv file_1 file_2**
Renames *file_1* as *file_2*.

rm Delete one or more files
e.g **rm my_file**
Deletes *my_file*.

ls List files in given directory (default is current directory)

cd Change to given directory (default is to home directory)
e.g **cd work**
Changes current directory to the subdirectory *work* if it exists.

pwd Print the full pathname of the current directory

mkdir Create one or more directories
e.g **mkdir work**
Creates the subdirectory *work* in the current directory if it does not already exist.

rmdir Delete the named directories
e.g **rmdir work**
Removes the subdirectory *work* of the current directory if it exists and is empty.

diff Generate report of lines that differ between two files
e.g **diff file_1 file_2**
Reports differences between *file_1* and *file_2*.

10 Text Filters

Note: all of the commands in this section (except for `ispell` output to **stdout** as opposed to a file - if you wish to save the output to a file, add

> *`output_file`*

to the end of the command.

wc Print a line, word and character count for the given file
e.g **wc my_file**
Generates statistics for file *my_file*.

ispell Interactive spelling checker
e.g **ispell my_file**
Runs through the file *my_file* for spell checking.

fmt Simple text formatter
e.g. **fmt input_file**
Will reproduce the file *input_file* with its lines as close as possible to 72 characters long, splitting them if necessary.

grep Search a file for a pattern
e.g. **grep jabberwocky my_file**
Will search the file *my_file* for the expression *jabberwocky* and report if it is contained therein.

head Display the first few lines of a file
e.g. **head -15 my_file**
Will display the first 15 lines of the file *my_file*. The default (i.e. `head my_file`) is to display the first 10 lines.

nl Line numbering filter
e.g. **nl input_file**
Will read lines from the file *input_file* and output them with numbers on the left.

sort Sort and/or merge files
e.g. **file_1 file_2**
Will sort all of the lines of files *file_1* and *file_2*.

tail Display last the few lines of a file
e.g. **tail -30 my_file**
Will display the last 30 lines of the file *my_file*. The default (i.e. `tail my_file`) is to display the last 10 lines.

11 Compilers & Debuggers

hugs The Hugs functional programming system
e.g. **hugs my_script.hs**
Evaluates the expressions contained in the file *my_script.hs*.

gcc GNU C Compiler
e.g. **gcc *options* my_prog.c**
Compiles the C program contained in the file *my_prog.c*, putting the executable output into *a.out*. Common *options* include **–O, –Wall, –g**, which specify compilation with **O**ptimization, **all W**arnings enabled and producing debugging information for use with **g**db respectively.

gnuc Runs gcc with almost all warnings enabled (not available on all systems).
e.g. **gnuc my_prog.c**
Compiles the C program contained in the file *my_prog.c*, strictly reporting warnings about the code. Puts the executable output into *a.out*.

gdb GNU Debugger
e.g. **gdb my_prog**
Runs **gdb** to examine the executable program contained in the file *my_prog*. For more detailed information, see the chapter on **gdb** in *Student Manual B*.

dbx A source-level debugger

12 Programming Utilities

error Insert compiler error messages at right source lines

13 On-line Manuals

man Display information from the on-line reference manuals
e.g. **man ls**
Displays information on the UNIX command **ls**.

apropos Locate commands by keyword lookup
e.g. **apropos copy**
Displays the names and summary lines of all manual pages which contain the string *copy* in their synopsis line. Same as **man –k**.

whatis Lookup one or more commands in the on-line manual pages
e.g. **whatis ls**
Displays information on what manual pages are available for UNIX command **ls**.

14 General Information

quota Display disk usage and limits
e.g. **quota -v**
Reports information on user's current disk usage and quota restrictions.

date Print the current date and time

15 Mail

elm Interactive mail system

mail The basic mail utility provided with UNIX. Simple, efficient, but not very user-friendly.
e.g. **mail**
Runs **mail** to read any mail the user has.
e.g. **mail foo**
Mails user *foo* whatever is typed next until **^D** is entered.

pine Program for Internet News and Email. Recommended.

16 News

pine Program for Internet News and Email. Recommended.

17 WWW

lynx A general purpose distributed information browser for the World Wide Web. Useful at dumb (non-graphical) terminals.

mozilla X Windows based graphical WWW browsing tool

18 Submitting Projects

submit Submit a project for assessment
e.g. **submit 171 A *file(s)***
Starts submission process for project *A* of subject *433-171*.

verify Verify that a previously submitted project has been received
e.g. **verify 171 A**
Displays the submitted material for project *A* of subject *433-171* with any output from running the program on test data.

Q vi Command Quick Reference

*n*SPACE, *n*BACKSPACE	Move *n* characters forward/backward
^ (SHIFT-6), $	Move to start/end of line
*n*RETURN, *n*-	Move to start of *n*th next/preceding line
*n*h, *n*j, *n*k, *n*l	Move *n* characters left/down/up/right
*n*w, *n*b	Move *n* words forward/backward
%	Go to matching bracket
*n*G	Move to line number *n* (default: last line)
H, M, L	Move to first/middle/last line of the screen
^D, ^U	Scroll forward/backward half a screen
^F, ^B	Move forward/backward one screen
/*string*/, ?*string*?	Search forward/backword for *string*
*n*cl, *n*cw, *n*cd	Change *n* chars/words/lines *
*n*dl, *n*dw, *n*dd	Delete *n* chars/words/lines
*n*s	Change *n* characters (cf. *n*dl)
rc	Replace current character by the character c
*n*x	Delete *n* characters (cf. *n*dl)
C	Change remainder of line *
D	Delete remainder of line
Y	Yank line
a, A	Append characters: after cursor/at end of line *
i, I	Insert characters: before cursor/at start of line *
o, O	Open line: below/above cursor *
p, P	put changed, deleted, yanked text before/after cursor
.	Repeat the most recent command
n	Repeat the most recent search command
u	Undo the most recent change to the editor buffer
^L	Redraw the screen
:w	Write editor buffer, overwriting the original file
:w *filename*	Write editor buffer to *filename* (overwriting if it exists)
:q	Quit vi
:q!	Quit vi without writing the editor buffer
:wq *or* ZZ	Write editor buffer (overwriting original file), then quit
:wq *filename*	Write editor buffer to *filename*, then quit

Notes:

1. The integer *n* preceding some of the above commands can be omitted, in which case the default value (= 1 unless otherwise noted) is used.
2. The commands SPACE, BACKSPACE and RETURN are typed by pressing the named key. The control commands (e.g., ^D) are typed by pressing the required key (e.g., d) *whilst holding down the* CONTROL *key*.
3. The colon commands (e.g. `:w`) and search commands (e.g. /*string*/) must be terminated by typing the RETURN key.
4. Commands marked * cause the editor to enter *insert* mode.
5. If you are uncertain what mode you are in, type the ESC key two or three times, thereby returning you to *command* mode.

R UNIX Troubleshooting

This section aims to give you some solutions to commonly occurring problems with UNIX. Please let your tutor know if you have any suggestions for additions to this section.

1 When I log out, the computer tells me: “There are stopped jobs”.

If this occurs, it may be because sometime during your session you (probably accidentally) pressed CONTROL-Z (`^Z`). This is a UNIX command that suspends (temporarily stops) a currently running command (or job). Any such *stopped* job should be either resumed to the *foreground* by using the command `fg`, or killed by entering the command; `kill %n`, where `n` is the number of the stopped job that is to be killed.

To identify the jobs that have been stopped, enter the command: `jobs`.

2 When I use vi, the screen gets all messed up.

If this happens, you probably have your terminal type set incorrectly. Enter the command:

```
set
```

and look at the line in the output which starts with “term”. The next word should be either “xterm” (if you are using the X-window system), or “vt100” (if you are using a dumb terminal). There are other possibilities: for some of the older dumb terminals, it should be “esprit”, and some terminal emulation programs that you use with modems will require other settings. (You will need to refer to the program’s documentation to find out what.) If you think your settings are incorrect, you can change it. Say you are on a dumb terminal, but it says “xterm”. To fix it, you would enter this command:

```
set term=vt100
```

3 Help! I am lost; I do not know how to get back to my home directory!

If you get lost when you’re exploring the directories, just enter the command:

```
cd
```

on its own to get back to your home directory.

4 When I try to erase characters, all I get is ^H or ^?

To remedy this problem, enter the following command at your UNIX prompt:

```
stty erase erase
```

where *erase* is the key that you want to use to delete characters.

If that key works at your UNIX prompt, but not when you are using `vi` or other programs, use the following command instead:

```
stty erase ^V erase
```

The `^V` will not appear on your screen.

5 The printer does not print my file!

This could be for a number of reasons, but the solution is to see a staff member in all cases. Firstly, check your print quota (ask a staff member what the command for this is).

If that is not the problem, go to the printer and see if it is jammed or if it is out of paper.

Some files cannot be printed directly using the `lpr` command. For example, Acrobat files must be printed from within Acrobat reader.