The unix programming environment

Edition 2.2, August 2001

Mark Burgess
Centre of Science and Technology
Faculty of Engineering, Oslo College

Foreword

This is a revised version of the UNIX compendium which is available in printed form and online via the WWW and info hypertext readers. It forms the basis for a one or two semester course in UNIX. The most up-to-date version of this manual can be found at

`http://www.iu.hio.no/~mark/unix/unix.html.`

It is a reference guide which contains enough to help you to find what you need from other sources. It is not (and probably can never be) a complete and self-contained work. Certain topics are covered in more detail than others. Some topics are included for future reference and are not intended to be part of an introductory course, but will probably be useful later. The chapter on X11 programming has been deleted for the time being.

Comments to Mark.Burgess@iu.hio.no Oslo, August 2001

Welcome

If you are coming to unix for the first time, from a Windows or MacIntosh environment, be prepared for a rather different culture than the one you are used to. UNIX is not about ‘products’ and off-the-shelf software, it is about open standards, free software and the ability to change just about everything.

- What you personally might perceive as user friendliness in other systems, others might perceive as annoying time wasting. UNIX offers you just about every level of friendliness and unfriendliness, if you choose your programs right. In this book, we take the programmer’s point of view.
- UNIX is about functionality, not about simplicity. Be prepared for powerful, not necessarily ‘simple’ solutions.

You should approach UNIX the way you should approach any new system: with an open mind. The journey begins...

Overview

In this manual the word "host" is used to refer to a single computer system -- i.e. a single machine which *has a name termed its "hostname".*

What is unix?

UNIX is one of the most important operating system in use today, perhaps even *the* most important. Since its invention around the beginning of the 1970s it has been an object of continual research and development. UNIX is not popular because it is the best operating system one could imagine, but because it is an extremely flexible system which is easy to extend and modify. It is an ideal platform for developing new ideas.

Much of the success of UNIX may be attributed to the rapid pace of its development (a development to which all of its users have been able to contribute) its efficiency at running programs and the many powerful tools which have been written for it over the years, such as the C programming language, `make`, shell, `lex` and `yacc` and many others. UNIX was written by programmers for programmers. It is popular in situations where a lot of computing power is required and for database applications, where timesharing is critical. In contrast to some operating systems, UNIX performs equally well on large scale computers (with many processors) and small computers which fit in your suitcase!

All of the basic mechanisms required of a multi-user operating system are present in UNIX. During the last few years it has become ever more popular and has formed the basis of newer, though less mature, systems like NT. One reason for this that now computers have now become powerful enough to run UNIX effectively. UNIX places burdens on the resources of a computer, since it expects to be able to run potentially many programs simultaneously.

If you are coming to UNIX from Windows or DOS you may well be used to using applications software or helpful interactive utilities to solve every problem. UNIX is not usually like this: the operating system has much greater functionality and provides the possibilities for making your own, so it is less common to find applications software which implements the same things. In UNIX you are usually asked to learn a language in order to express exactly what you want. This is much more powerful than menu systems, but it is harder to learn

UNIX has long been in the hands of academics who are used to making their own applications or writing their own programs, whereas as the Windows world has been driven by businesses who are willing to spend money on software. For that reason commercial UNIX software is often very expensive and therefore not available at this college. On the other hand, the flexibility of UNIX means that it is easy to write programs and it is possible to fetch gigabytes of *free software* from the Internet to suit your needs. It may not look exactly like what you are used to on your PC, but then you have to remember that UNIX users are a different kind of animal altogether

Like all operating systems, UNIX has many faults. The biggest problem for any operating system is that it evolves without being redesigned. Operating systems evolve as more and more patches and hacks are applied to solve day-to-day problems. The result is either a mess which works somehow (like UNIX) or a blank refusal to change (like DOS or the MacIntosh, prior to MacOS X, which is based on BSD UNIX). From a practical perspective, UNIX is important and successful because it is a multi-process system which

- has an enormous functionality built in, and the capacity to adapt itself to changing technologies,
- is relatively portable,

- is good at sharing resources (but not so good at security),
- has tools which are each developed to do *one thing well*,
- allows these tools to be combined in every imaginable way, using pipes and channeling of data streams,
- incorporates networking almost trivially, because all the right mechanisms are already there for providing services and sharing, building client-server pairs etc,.
- it is very adaptable and is often used to develop new ideas because of the rich variety of tools it possesses.

UNIX has some problems: it is old, it contains a lot of rubbish which no one ever bothered to throw away. Although it develops quickly (at light speed compared to either DOS/Windows or MacIntosh) the user interface has been the slowest thing to change. UNIX is not user-friendly for beginners, it is user-friendly for advanced users: it is made for users who *know* about computing. It sometimes makes simple things difficult, but above all it makes things possible!

The aim of this introduction is to

- introduce the unix system basics and user interface,
- develop the unix philosophy of using and combining tools,
- learn how to make new tools and write software,
- learn how to understand existing software.

To accomplish this task, we must first learn something about the shell language (the way in which UNIX starts programs). Later we shall learn how to solve more complex problems using Perl and C. Each of these is a language which can be used to put UNIX to work. We must also learn when to use which tool, so that we do not waste time and effort. Typical uses for these different interfaces are

shell Command line interaction, making scripts which performs simple jobs such as running programs, installing new software, simple system configuration and administration.

perl Text interpretation, text formatting, output filters, mail robots, WWW cgi (common gateway interface) scripts in forms, password testing, simple database manipulation, simple client-server applications.

C Nearly all of UNIX is written in C. Any program which cannot be solved quickly using shell or perl can be written in C. One advantage is that C is a compiled language and many simple errors can be caught at compile time.

Much of UNIX's recent popularity has been a result of its networking abilities: UNIX is the backbone of the Internet. No other widely available system could keep the Internet alive today. GNU/Linux is a free/open source re-write of the UNIX operating system, which many enhancements. While GNU/Linux is not "rocket science" to computer experts, it has distilled the essence of UNIX and placed it in the hands of everyone. It runs on wrist watches and mainframe computers. Like it or loathe it, GNU/Linux is probably the most important single development in computer operating systems for many years.

Once you have mastered the UNIX interface and philosophy you will find that i) the PC and MacIntosh window environments might seem to be easy to use, but are simplistic and primitive by comparison; ii) UNIX is far from being the perfect operating system--it has a whole different set of problems and flaws.

The operating system of the future will not be UNIX or GNU/Linux as we see it today (hopefully), nor will is be DOS or MacIntosh, but one thing is for certain: it will owe a lot to the UNIX operating system and will contain many of the tools and mechanisms we shall describe below.

Flavours of unix

UNIX is not a single operating system. It has branched out in many different directions since it was introduced by AT&T. The most important `'fork()'` in its history happened early on when the university of Berkeley, California created the BSD (Berkeley Software Distribution), adding network support and the C-shell.

Here are some of the most common implementations of unix.

BSD:
: Berkeley, BSD

SunOS:
: Sun Microsystems, BSD/sys 5

Solaris:
: Sun Microsystems, Sys 5/BSD

Ultrix:
: Digital Equipment Corporation, BSD

OSF 1:
: Digital Equipment Corporation, BSD/sys 5

HPUX:
: Hewlett-Packard, Sys 5

AIX: IBM, Sys 5 / BSD

IRIX:
: Silicon Graphics, Sys 5

GNU/Linux:
: GNU, BSD/Posix

How to use this reference guide

This programming guide is something between a user manual and a tutorial. The information contained here should be sufficient to get you started with the unix system, but it is far from complete.

To use this programming guide, you will need to work through the basics from each chapter. You will find that there is much more information here than you need straight away, so try not to be overwhelmed by the amount of material. Use the contents and the indices at the back to find the information you need. If you are following a one-semester UNIX course, you should probably concentrate on the following:

- The remainder of this introduction
- The detailed knowledge of the Bash shell
- A detailed knowledge of Perl, guided by chapter 6. This chapter provides pointers on how to get started in perl. It is not a substitute for the perl book.
- A sound appreciation of chapter 8 on network programming.

The *only* way to learn UNIX is to sit down and try it. As with any new thing, it is a pain to get started, but once you are started, you will probably come to agree that UNIX contains a wealth of possibilities, perhaps more than you had ever though was possible or useful!

One of the advantages of the UNIX system is that the entire UNIX manual is available on-line. You should get used to looking for information in the online manual pages. For instance, suppose you do not remember how to create a new directory, you could do the following:

```
nexus% man -k dir

dir             ls (1)          - list contents of directories
dirname         dirname (1)     - strip non-directory suffix from file name
dirs            bash (1)        - bash built-in commands, see bash(1)
find            find (1)        - search for files in a directory hierarchy
ls              ls (1)          - list contents of directories
mkdir           mkdir (1)       - make directories
pwd             pwd (1)         - print name of current/working directory
rmdir           rmdir (1)       - remove empty directories
```

The `'man -k'` command looks for a keyword in the manual and lists all the references it finds. The command `'apropos'` is completely equivalent to `'man -k'`. Having discovered that the command to create a directory is `'mkdir'` you can now look up the specific manual page on `'mkdir'` to find out how to use it:

```
man mkdir
```

Some but no all of the UNIX commands also have a help option which is activated with the `'-h'` or `'--help'` command-line option.

```
dax% mkdir --help
Usage: mkdir [OPTION] DIRECTORY...

  -p, --parents     no error if existing, make parent directories as needed
  -m, --mode=MODE   set permission mode (as in chmod), not 0777 - umask
      --help        display this help and exit
      --version     output version information and exit
dax%
```

NEVER-DO's in UNIX

There are some things that you should never do in UNIX. Some of these will cause you more serious problems than others. You can make your own list as you discover more.

- You should NEVER EVER switch off the power on a UNIX computer unless you know what you are doing. A UNIX machine is not like a PC running DOS. Even when you are not doing anything, the system is working in the background. If you switch off the power, you could interrupt the system while it is writing to the disk drive and destroy your disk. You must also remember that several users might be using the system even though you cannot see them: they do not have to be sitting at the machine, they could be logged in over the network. If you switch off the power, you might ruin their valuable work.
- Once you have deleted a UNIX file using `rm` it is impossible to recover it! Don't use wildcards with `rm` without thinking quite carefully about what you are doing! It has happened to very many users throughout the history of UNIX that one tries to type

```
rm *~
```

but instead, by a slip of the hand, one writes

```
rm * ~
```

UNIX then takes these wildcards in turn, so that the first command is `rm *` which deletes all of your files! BE CAREFUL!

- Don't ever call a program or an important file `'core'`. Many scripts go around deleting files called `'core'` because the, when a program crashes, UNIX dumps the entire kernel image to a file called `'core'` and these files use up a lot of disk space. If you call a file `'core'` it might get deleted!
- Don't call test programs `test`. There is a UNIX command which is already called test and chances are that when you try to run your program you will start the UNIX command instead. This can cause a lot of confusion because the UNIX command doesn't seem to do very much at all!

What you should know before starting

One library: several interfaces

The core of unix is the library of functions (written in C) which access the system. Everything you do on a unix system goes through this set of functions. However, you can choose your own interface to these library functions. UNIX has very many different interfaces to its libraries in the form of languages and command interpreters.

You can use the functions directly in C, or you can use command programs like `'ls'`, `'cd'` etc. These functions just provide a simple user interface to the C calls. You can also use a variety of 'script' languages: C-shell, Bourne shell, Perl, Tcl, scheme. You choose the interface which solves your problem most easily.

UNIX commands are files

With the exception of a few simple commands which are built into the command interpreter (shell), all unix commands and programs consist of executable files. In other words, there is a separate executable file for each command. This makes it extremely simple to add new commands to the system. One simply makes a program with the desired name and places it in the appropriate directory.

UNIX commands live in special directories (usually called `bin` for binary files). The location of these directories is recorded in a variable called `path` or `PATH` which is used by the system to search for binaries. We shall return to this in more detail in later chapters.

Kernel and Shell

Since users cannot command the kernel directly, UNIX has a command language known as the *shell*. The word shell implies a layer around the kernel. A shell is a user interface, or command interpreter.

There are two main versions of the shell, plus a number of enhancements.

/bin/sh
: The Bourne Shell. The shell is most often used for writing system scripts. It is part of the original unix system.

/bin/csh
: The C-shell. This was added to unix by the Berkeley workers. The commands and syntax resemble C code. C-shell is better suited for interactive work than the Bourne shell.

The program `tcsh` is a public-domain enhancement of the csh and is in common use. Two improved versions of the Bourne shell also exist: `ksh`, the Korn shell and `bash`, the Bourne-again shell.

Although the shells are mainly tools for typing in commands (which are executable files to be loaded and run), they contain features such as aliases, a command history, wildcard-expansions and job control functions which provide a comfortable user environment.

The role of C

Most of the unix kernel and daemons are written in the C programming language (1). Calls to the kernel and to services are made through functions in the standard C library. The commands like `chmod`, `mkdir` and `cd` are all C functions. The binary files of the same name `/bin/chmod`, `/bin/mkdir` etc. are just trivial "wrapper" programs for these C functions.

Until *Solaris 2*, the C compiler was a standard part of the UNIX operating system, thus C is the most natural language to program in in a UNIX environment. Some tools are provided for C programmers:

dbx A symbolic debugger. Also *gdb*, *xxgdb ddd*.

make
: A development tool for compiling large programs.

lex A ‘lexer’. A program which generates C code to recognize words of text.

yacc A ‘parser’. This is a tool which generates C code for checking the syntax of groups of textual words.

rpcgen
: A protocol compiler which generates C code from a higher level language, for programming RPC applications.

Stdin, stdout, stderr

UNIX has three logical *streams* or *files* which are always open and are available to any program.

stdin The standard input - file descriptor 0.

stdout
: The standard output - file descriptor 1.

stderr
: The standard error - file descriptor 2.

The names are a part of the C language and are defined as pointers of type `FILE`.

```
#include <stdio.h>
```

```
/* FILE *stdin, *stdout, *stderr; */

fprintf(stderr,"This is an error message!\n");
```

The names are ‘logical’ in the sense that they do not refer to a particular device, or a particular place for information to come from or go. Their role is analogous to the ‘.’ and ‘..’ directories in the filesystem. Programs can write to these files without worrying about where the information comes from or goes to. The user can personally define these places by *redirecting standard I/O*. This is discussed in the next chapter.

A separate stream is kept for error messages so that error output does not get mixed up with a program’s intended output.

The superuser (root) and *nobody*

When logged onto a UNIX system directly, the user whose name is `root` has unlimited access to the files on the system. `root` can also become any other user without having to give a password. `root` is reserved for the system administrator or *trusted users*.

Certain commands are forbidden to normal users. For example, a regular user should not be able to halt the system, or change the ownership of files (see next paragraph). These things are reserved for the `root` or *superuser*.

In a networked environment, `root` has no automatic authority on remote machines. This is to prevent the system administrator of one machine in Canada from being able to edit files on another in China. He or she must log in directly and supply a password in order to gain access privileges. On a network where files are often accessible in principle to anyone, the username `root` gets mapped to the user `nobody`, who has no rights at all.

The file hierarchy

UNIX has a hierarchical filesystem, which makes use of directories and sub-directories to form a tree. The root of the tree is called the root filesystem or ‘/’. Although the details of where every file is located differ for different versions of unix, some basic features are the same. The main sub-directories of the root directory together with the most important file are shown in the figure. Their contents are as follows.

`‘/bin’`
: Executable (binary) programs. On most systems this is a separate directory to /usr/bin. In SunOS, this is a pointer (link) to /usr/bin.

`‘/etc’`
: Miscellaneous programs and configuration files. This directory has become very messy over the history of UNIX and has become a dumping ground for almost anything. Recent versions of unix have begun to tidy up this directory by creating subdirectories `‘/etc/mail’`, `‘/etc/services’` etc!

`‘/usr’`
: This contains the main meat of UNIX. This is where application software lives, together with all

of the basic libraries used by the OS.

`‘/usr/bin’`
: More executables from the OS.

`‘/usr/local’`
: This is where users’ custom software is normally added.

`‘/sbin’`
: A special area for statically linked system binaries. They are placed here to distinguish commands used solely by the system administrator from user commands and so that they lie on the system root partition where they are guaranteed to be accessible during booting.

`‘/sys’`
: This holds the configuration data which go to build the system kernel. (See below.)

`‘/export’`
: Network servers only use this. This contains the disk space set aside for client machines which do not have their own disks. It is like a ‘virtual disk’ for diskless clients.

`‘/dev, /devices’`
: A place where all the ‘logical devices’ are collected. These are called ‘device nodes’ in unix and are created by `mknod`. Logical devices are UNIX’s official entry points for writing to devices. For instance, `/dev/console` is a route to the system console, while `/dev/kmem` is a route for reading kernel memory. Device nodes enable devices to be treated as though they were files.

`‘/home’`
: (Called */users* on some systems.) Each user has a separate login directory where files can be kept. These are normally stored under `/home` by some convention decided by the system administrator.

`‘/var’`
: System 5 and mixed systems have a separate directory for spooling. Under old BSD systems, `/usr/spool` contains spool queues and system data. `/var/spool` and `/var/adm` etc are used for holding queues and system log files.

`‘/vmunix’`
: This is the program code for the unix *kernel* (see below). On HPUX systems with file is called `‘hp-ux’`. On linux it is called `‘linux’`.

`‘/kernel’`
: On newer systems the kernel is built up from a number of modules which are placed in this directory.

Every unix directory contains two ‘virtual’ directories marked by a single dot and two dots.

```
ls -a
.   ..
```

The single dot represents the directory one is already in (the current directory). The double dots mean the directory one level up the tree from the current location. Thus, if one writes

```
cd /usr/local
cd ..
```

the final directory is `/usr`. The single dot is very useful in C programming if one wishes to read ‘the current directory’. Since this is always called ‘.’ there is no need to keep track of what the current directory really is.

‘.’ and ‘..’ are ‘hard links’ to the true directories.

Symbolic links

A symbolic link is a pointer or an alias to another file. The command

```
ln -s fromfile /other/directory/tolink
```

makes the file `fromfile` appear to exist at `/other/directory/tolink` simultaneously. The file is not copied, it merely appears to be a part of the file tree in two places. Symbolic links can be made to both files and directories.

A symbolic link is just a small file which contains the name of the real file one is interested in. It cannot be opened like an ordinary file, but may be read with the C call `readlink()` See section lstat and readlink. If we remove the file a symbolic link points to, the link remains -- it just points nowhere.

Hard links

A *hard link* is a duplicate *inode* in the filesystem which is in every way equivalent to the original file inode. If a file is pointed to by a hard link, it cannot be removed until the link is removed. If a file has @math{n} hard links -- all of them must be removed before the file can be removed. The number of hard links to a file is stored in the filesystem *index node* for the file.

Getting started

If you have never met unix, or another multiuser system before, then you might find the idea daunting. There are several things you should know.

Logging in

Each time you use unix you must log on to the system by typing a username and a password. Your login name is sometimes called an ‘account’ because some unix systems implement strict quotas for computer resources which have to be paid for with real money(2).

```
  login: mark
  password:
```

Once you have typed in your password, you are ‘logged on’. What happens then depends on what kind of system you are logged onto and how. If you have a colour monitor and keyboard in front of you, with a graphical user interface, you will see a number of windows appear, perhaps a menu bar. You then use a mouse and keyboard just like any other system.

This is not the only way to log onto unix. You can also log in remotely, from another machine, using the Secure Shell `ssh` program (this replaces the now antiquated `telnet` and `rlogin` programs). If you use these programs, you will normally only get a text or command line interface (though graphical interfaces can easily be arranged).

Once you have logged in, a short message will be printed (called Message of the Day or motd) and you will see the C-shell prompt: the name of the host you are logged onto followed by a percent sign, e.g.

```
Linux cube 2.2.19pre13 #2 Mon Feb 26 15:53:31 MET 2001 i686 unknown

   This is GNU/Linux - send problems to help@example.org

10:44pm  up 8 days, 13:34,  3 users,  load average: 0.08, 0.02, 0.01

There are 480 messages in your incoming mailbox.
```

Remember that every UNIX machine is a separate entity: it is not like logging onto a PC system where you log onto the ‘network’ i.e. the PC file server. Every UNIX machine is a server, or a client -- more correctly a "peer" (equal partner). The network, in unix-land, has lots of players.

The first thing you should do once you have logged on is to set a reliable password. A poor password might be okay on a PC which is not attached to a large network, but once you are attached to the Internet, you have to remember that the whole world will be trying to crack your password. Don’t think that no one will bother: some people really have nothing better to do. A password should not contain any word that could be in a list of words (in any language), or be a simple concatenation of a word and a number (e.g. mark123). It takes seconds to crack such a password. Choose instead something which is easy to remember. Feel free to use the PIN number from your bankers card in your password! This will leave you with fewer things to remember. e.g. Ma9876rk). Passwords can be up to eight characters long.

Some sites allow you to change your password anywhere. Other sites require you to log onto a special machine to change your password:

```
 dax%
 dax% passwd
 Change your password on host nexus
 You cannot change it here
 dax% rlogin nexus
 password: ******

 nexus% passwd
 Changing password for mark
 Enter login password: ********
 Enter new password: ********
 Reenter new passwd: ********
```

You will be prompted for your old password and your new password twice. If your network is large, it might take the system up to an hour or two to register the change in your password, so don’t forget the old one right away!

Mouse buttons

UNIX has three mouse buttons. On some PC’s running GNU/Linux or some other PC unix, there are only two, but the middle mouse button can be simulated by pressing both mouse buttons simultaneously. The mouse buttons have the following general functions. They may also have additional functions in special software.

index finger
: This is used to select and click on objects. It is also used to mark out areas and copy by dragging. This is the button you normally use.

middle finger
: Used to pull down menus. It is also used to paste a marked area somewhere at the mouse position.

outer finger
: Pulls down menus.

On a left-handed system right and left are reversed.

E-mail

Reading electronic mail on unix is just like any other system, but there are many programs to choose from. There are very old programs from the seventies such as

```
mail
```

and there are fully graphical mail programs such as

```
tkrat
mailtool
```

Choose the program you like best. Not all of the programs support modern multimedia extensions because of their age. Some programs like `tkrat` have immediate mail notification alerts. To start a mail program you just type its name. If you have an icon-bar, you can click on the mail-icon.

Simple commands

Inexperienced computer users often prefer to use file-manager programs to avoid typing anything. With a mouse you can click your way through directories and files without having to type anything (e.g. the `kfm` or `tkdesk` programs). More experienced users generally find this to be slow and tedious after a while and prefer to use written commands. UNIX has many short cuts and keyboard features which make typed commands extremely fast and much more powerful than use of the mouse.

Today the CDE, KDE and GNOME projects are the most important efforts to write graphical user interfaces for computers. The CDE (Common Desktop Environment) is a commercial program developed by IBM, Hewlett-Packard, Sun Microsystems and many other vendors. KDE (a German effort, a pun on CDE) and GNOME are free software window systems which have taken windowing to the next level. While they have borrowed and stolen many ideas from Windows' innovative Windows 95 user interface, they have taken windowing beyond this.

If you come from a Windows environment, the UNIX commands can be a little strange. It is a different way of thinking: using language to ask for exactly what you want, instead of pointing to a menu of limited choices. It is also a strange language. Because they stem from an era when keyboards had to be hit with hammer force, and machines were very slow, the UNIX command names are as short as possible, so they seem pretty cryptic. Some familiar ones which DOS borrowed from UNIX include,

```
cd
mkdir
```

which change to a new directory and make a new directory respectively. To list the files in the current directory you use,

```
ls
```

To rename a file, you ‘move’ it:

```
mv old-name new-name
```

Text editing and word processing

Text editing is one of the things which people spend most time doing on any computer. It is important to distinguish text editing from word processing. On a PC or MacIntosh, you are perhaps used to Word or WordPerfect for writing documents.

UNIX has a Word-like program called `lyx`, and even several Office clones (e.g. Star Office `soffice`), but for the most part UNIX users do not use word processors. It is more common in the UNIX community to write all documents, regardless of whether they are letters, books or computer programs, using a non-formatting text editor. (UNIX word processors like `Framemaker` do exist, but they are very expensive. A version of MS-Word also exists for some unices.) Once you have written a document in a normal text editor, you call up a text formatter to make it pretty. You might think this strange, but the truth of the matter is that this two-stage process gives you the most power and flexibility--and that is what most UNIX folks like.

For writing programs, or anything else, you edit a file by typing:

```
 emacs myfile
```

`emacs` is one of dozens of text-editors. It is not the simplest or most intuitive, but it is the most powerful and if you are going to spend time learning an editor, it wouldn’t do any harm to make it this one. You could also click on emacs’ icon if you are relying on a window system. Emacs is almost certainly the most powerful text editor that exists on any system. It is not a word-processor, it is not for formatting printed documents, but it can be linked to almost any other program in order to format and print text. It contains a powerful programming language and has many intelligent features. We shall not go into the details of document formatting in this book, but only mention that programs like `troff` and `Tex` or `Latex` are used for this purpose to obtain typeset-quality printing. Text formatting is an area where UNIX folks do things differently to PC folks.

The login environment

UNIX began as a timesharing mainframe system in the seventies, when the only terminals available were text based *teletype* terminals or *tty*-s. Later, the Massachusetts Institute of Technology (MIT) developed the X-windows interface which is now a standard across UNIX platforms. Because of this history, the X-window system works as a front end to the standard UNIX shell and interface, so to understand the user environment we must first understand the shell.

Shells

A shell is a command interpreter. In the early days of UNIX, a shell was the only way of issuing commands to the system. Nowadays many window-based application programs provide menus and buttons to perform simple commands, but the UNIX shell remains the most powerful and flexible way of interacting with the system.

After logging in and entering a password, the UNIX process *init* starts a shell for the user logging in. UNIX has several different kinds of shell to choose from, so that each user can pick his/her favourite command interface. The type of shell which the system starts at login is determined by the user's entry in the *passwd* database. On most systems, the standard login shell is a variant of the C-shell.

Shells provide facilities and commands which

- Start and stop processes (programs)
- Allow two processes to communicate through a *pipe*
- Allow the user to redirect the flow of input or output
- Allow simple command line editing and command history
- Define aliases to frequently used commands
- Define global "environment" variables which are used to configure the default behaviour of a variety of programs. These lie in an "associated array" for each process and may be seen with the `'env'` command. Environment variables are inherited by all processes which are started from a shell.
- Provide wildcard expansion (joker notation) of filenames using `'*,?,[]'`
- Provide a simple script language, with tests and loops, so that users can combine system programs to create new programs of their own.
- Change and remember the location of the current working directory, or location within the file hierarchy.

The shell does not contain any more specific functions--all other commands, such as programs which list files or create directories etc., are executable programs which are independent of the shell. When you type `'ls'`, the shell looks for the executable file called `'ls'` in a special list of directories called *the command path* (which is contained in the environment variable $PATH) and attempts to start this program. This allows such programs to be developed and replaced independently of the actual command interpreter.

Each shell which is started can be customized and configured by editing a setup file. For the Bash shell this file is called `'.bashrc'`, and for the C-shell and its variants it is called `'.profile'`. (Note that files which begin with leading dots are not normally visible with the `'ls'` command. Use `'ls -a'` to view these.) Any commands which are placed in these files are interpreted by the shell before the first command prompt is issued. These files are typically used to define a command search path and terminal characteristics.

On each new command line you can use the cursor keys to edit the line. The up-arrow browses back through earlier commands. `CTRL-a` takes you to the start of the line. `CTRL-e` takes you to the end of the line. The `TAB` can be used to save typing with the 'completion' facility See section Command/filename *completion.*

Shell commands generally

Shell commands are commands like `cp`, `mv`, `passwd`, `cat`, `more`, `less`, `cc`, `grep`, `ps` etc..

One thing you can always bet on with Unix is that there is not just one way of doing things -- there are so many standards, that there is often a bewildering array to choose from. UNIX has two main command shells. They are called `sh` (Bourne Shell) and `csh` C-shell. Their modern implementations are called `Bash` (Bourne Again Shell) and `tcsh` (T-C shell).

Very few commands are actually built into the shell command line interpreter, in the same way that they are built into DOS. Rather commands are programs which exist as actual program files. When we type a command, the shell searches for a program with the same name and tries to execute it. This is very flexible, since anyone is free to write their own programs and therefore extend the command language of the system. The file must be executable, or a `Command not found` error will result. To see what actually happens when you type a command like `gcc`, try typing the following into a GNU/Linux system: (you can type this exactly as shown into a Bash shell)

```
cube$ IFS=:

cube$ for dir in $PATH       # for every directory in the list path
>do
>  if [ -x $dir/gcc ]        # if the file is executable
>  then
>     echo Found $dir/gcc    # Print message found!
>     break                  # break out of loop
>  else
>     echo Searching $dir/gcc
>  fi
>done
```

If you use C-shell (e.g. tcsh), try typing in the following C-shell commands *directly into a C-shell.*

```
nexus%  foreach dir ( $path )  # for every directory in the list path
>    if ( -x $dir/gcc ) then   # if the file is executable
>      echo Found $dir/gcc     # Print message found!
>      break                   # break out of loop
>    else
>      echo Searching $dir/gcc
>    endif
>  end
```

The output of these command sequences is something like this:

```
  Searching /usr/lang/gcc
  Searching /usr/openwin/bin/gcc
  Searching /usr/openwin/bin/xview/gcc
  Searching /physics/lib/framemaker/bin/gcc
  Searching /physics/motif/bin/gcc
  Searching /physics/mutils/bin/gcc
  Searching /physics/common/scripts/gcc
  Found /physics/bin/gcc
```

If you type

```
  echo $PATH
```

in Bourne Shell, or

```
  echo $path
```

in C-shell you will see the entire list of directories which are searched by the shell. If we had left out the 'break' command, we might have discovered that UNIX often has several programs with the same name, in different directories! For example,

```
/bin/mail
/usr/ucb/mail
/bin/Mail

/bin/make
/usr/local/bin/make.
```

Also, different versions of UNIX have different conventions for placing the commands in directories, so the path list needs to be different for different types of UNIX machine. In Bash a few basic commands like `cd` and `kill` are built into the shell (as in DOS).

You can find out which directory a command is stored in using

```
type
```

command. For example

```
cube$ type cd
cd is a shell builtin
cube$ type mv
mv is /bin/mv
cube$
```

`type` only searches the directories in `$PATH` and quits after the first match, so if there are several commands with the same name, you will only see the first of them using `type`.

Finally, in the C-shell the command corresponding to type is built in and called `which`. In Bash `which` is a program:

```
cube$ type which
which is /usr/bin/which
cube$ tcsh
cube% which which
which: shell built-in command.
```

Take a look at the script `/usr/bin/which`. It is a script written in bash.

Environment and shell variables

Environment variables are variables which the shell keeps. They are normally used to configure the behaviour of utility programs like `lpr` (which sends a file to the printer) and `mail` (which reads and sends mail) so that special options do not have to be typed in every time you run these programs.

Any program can read these variables to find out how you have configured your working environment. We shall meet these variables frequently. Here are some important variables

```
PATH             # The search path for shell commands (bash)
TERM             # The terminal type (bash and csh)
DISPLAY          # X11 - the name of your display
LD_LIBRARY_PATH  # Path to search for object and shared libraries
HOSTNAME         # Name of this UNIX host
PRINTER          # Default printer (lpr)
HOME             # The path to your home directory (bash)
PS1              # The default prompt for bash

path             # The search path for shell commands (csh)
term             # The terminal type (csh)
prompt           # The default prompt for csh
home             # The path to your home directory (csh)
```

These variables fall into two groups. Traditionally the first group always have names in uppercase letters and are called *environment variables*, whereas variables in the second group have names with lowercase letters and are called *shell variables*-- but this is only a convention. The uppercase variables are *global variables*, whereas the lower case variables are *local variables*. Local variables are not defined for programs or sub-shells started by the current shell, while global variables are inherited by all sub-shells.

The Bash-shell and the C-shell use these conventions differently and not always consistently. You will see how to define these below. For now you just have to know that you can use the command `env` can be used in Bash shell to see all of the defined global environment variables while `set` lists both the global and the local variables.

Wildcards

Sometimes you want to be able to refer to several files in one go. For instance, you might want to copy all files ending in '.c' to a new directory. To do this one uses *wildcards*. Wildcards are characters like `*` `?` which stand for any character or group of characters. In card games the joker is a 'wild card' which can be substituted for any other card. Use of wildcards is also called *filename substitution* in the UNIX manuals, in the sections on `sh` and `csh`.

The wildcard symbols are,

`'?'` Match single character. e.g. `ls /etc/rc.????`
`'*'` Match any number of characters. e.g. `ls /etc/rc.*`
`'[...]'`
  Match any character in a list enclosed by these brackets. e.g. `ls [abc].C`

Here are some examples and explanations.

`'/etc/rc.????'`
  Match all files in /etc whose first three characters are `rc.` and are 7 characters long.
`'*.c'`

Match all files ending in ‘.c’ i.e. all C programs.

`‘*.[Cc]’`

List all files ending on ‘.c’ or ‘.C’ i.e. all C and C++ programs.

`‘*.[a-z]’`

Match any file ending in .a, .b, .c, ... up to .z etc.

It is important to understand that *the shell expands wildcards*. When you type a command, the program is not invoked with an argument that contains `*` or `?`. The shell expands the special characters first and invokes commands with the entire list of files which match the patterns. The programs never see the wildcard characters, only the list of files they stand for. To see this in action, you can type

```
echo /etc/rc*
```

which gives

```
/etc/rc0 /etc/rc0.d /etc/rc1 /etc/rc1.d /etc/rc2 /etc/rc2.d /etc/rc3
/etc/rc3.d /etc/rc5 /etc/rc6 /etc/rcS /etc/rcS.d
```

All shell commands are invoked with a command line of this form. This has an important corollary. It means that multiple renaming *cannot work*!

UNIX files are renamed using the `mv` command. In many microcomputer operating systems one can write

```
rename *.x *.y
```

which changes the file extension of all files ending in ‘.x’ to the same name with a ‘.y’ extension. This cannot work in UNIX, because the shell tries expands everything before passing the arguments to the command line.

Regular expressions

The wildcards belong to the shell. They are used for matching filenames. UNIX has a more general and widely used mechanism for matching *strings*, this is through *regular expressions*.

Regular expressions are used by the `egrep` utility, text editors like `ed`, `vi` and `emacs` and `sed` and `awk`. They are also used in the C programming language for matching input as well as in the Perl programming language and `lex` tokenizer. Here are some examples using the `egrep` command which print lines from the file `/etc/rc` which match certain conditions. The construction is part of `egrep`. Everything in between these symbols is a regular expression. Notice that special shell symbols `! * &` have to be preceded with a backslash `\` in order to prevent the shell from expanding them!

```
# Print all lines beginning with a comment #
egrep '(^#)'           /etc/rc

# Print all lines which DON'T begin with #
egrep '(^[^#])'        /etc/rc

# Print all lines beginning with e, f or g.
egrep '(^[efg])'       /etc/rc
```

```
# Print all lines beginning with uppercase
egrep '(^[A-Z])'       /etc/rc

# Print all lines NOT beginning with uppercase
egrep '(^[^A-Z])'      /etc/rc

# Print all lines containing ! * &
egrep '([\!\*\&])'     /etc/rc

# All lines containing ! * & but not starting #
egrep '([^#][\!\*\&])' /etc/rc
```

Regular expressions are made up of the following 'atoms'.

These examples assume that the file `'/etc/rc'` exists. If it doesn't exist on the machine you are using, try to find the equivalent by, for instance, replacing `/etc/rc` with `/etc/rc*` which will try to find a match beginning with the rc.

`'.'`	Match any single character except the end of line.
`'^'`	Match the beginning of a line as the first character.
`'$'`	Match end of line as last character.
`'[..]'`	Match any character in the list between the square brackets.(see below).
`'*'`	Match zero or more occurrences of the preceding expression.
`'+'`	Match one or more occurrences of the preceding expression.
`'?'`	Match zero or one occurrence of the preceding expression.

You can find a complete list in the UNIX manual pages. The square brackets above are used to define a *class* of characters to be matched. Here are some examples,

- If the square brackets contain a list of characters, $[a-z156]$ then a single occurrence of any character in the list will match the regular expression: in this case any lowercase letter or the numbers 1, 5 and 6.
- If the first character in the brackets is the caret symbol `'^'` then any character *except* those in the list will be matched.
- Normally a dash or minus sign `'-'` means a range of characters. If it is the first character after the `'['` or after `'[^'` then it is treated literally.

Nested shell commands and "

The backwards apostrophes '...' can be used in all shells and also in the programming language Perl. When these are encountered in a string the shell tries to execute the command inside the quotes and replace the quoted expression by the result of that command. For example:

```
UNIX$ echo "This system's kernel type is '/usr/bin/file /boot/vmlinuz-2.2.19pre13'"
This system's kernel type is /boot/vmlinuz-2.2.19pre13: Linux kernel x86 boot execut

UNIX$ for file in 'ls /local/ssl/misc/*'
> do
> echo I found a config file $file
> echo Its type is '/usr/bin/file $file'
> done
I found a config file /local/ssl/misc/CA.pl
Its type is /local/ssl/misc/CA.pl: perl script text
I found a config file /local/ssl/misc/CA.sh
Its type is /local/ssl/misc/CA.sh: Bourne shell script text
I found a config file /local/ssl/misc/c_hash
Its type is /local/ssl/misc/c_hash: Bourne shell script text
I found a config file /local/ssl/misc/c_info
Its type is /local/ssl/misc/c_info: Bourne shell script text
I found a config file /local/ssl/misc/c_issuer
Its type is /local/ssl/misc/c_issuer: Bourne shell script text
I found a config file /local/ssl/misc/c_name
Its type is /local/ssl/misc/c_name: Bourne shell script text
I found a config file /local/ssl/misc/der_chop
Its type is /local/ssl/misc/der_chop: perl script text
```

This is how we insert the result of a shell command into a text string or variable.

UNIX command overview

Important keys

`TAB` The `TAB` key is used by Bash and Emacs for "filename completion", i.e. when you are uncertain of the correct name of something, or simply can't be bothered to type it out, you can hit `TAB` to either finish off the word, or show you alternative choices. e.g. try in Bash

```
cube$ loadTAB
loadkeys    loadmeter   loadunimap
```

This shows the possible completions of commands which match "load". Type one more letter and `TAB`, and the rest will be filled in.

`CTRL-A`
: Jump to start of line. If `'screen'` is active, this prefixes all control key commands for `'screen'` and then the normal `CTRL-A` is replaced by `CTRL-a a`.

`CTRL-C`
: Interrupt or break key. Sends signal 15 to a process.

`CTRL-D`
: Signifies `'EOF'` (end of file) or shows expansion matches in command/filename completion See section Command/filename completion.

`CTRL-E`
: Jump to end of line.

`CTRL-L`
: Clear screen in newer shells and in emacs. Same as `'clear'` in the shell.

`CTRL-Z`
: Suspend the present process, but do not destroy it. This sends signal 18 to the process.

Alternative shells

`bash`	The Bourne Again shell, an improved sh.
`csh`	The standard C-shell.
`jsh`	The same as sh, with C-shell style job control.

`ksh` The Korn shell, an improved sh.
`sh` The original Bourne shell.
`sh5` On ULTRIX systems the standard Bourne shell is quite stupid. sh5 corresponds to the normal Bourne shell on these systems.
`tcsh` An improved C-shell.
`zsh` An improved sh.

Window based terminal emulators

`xterm`
: The standard X11 terminal window.

`shelltool, cmdtool`
: Openwindows terminals from Sun Microsystems. These are not completely X11 compatible during copy/paste operations.

`screen`
: This is not a window in itself, but allows you to emulate having several windows inside a single (say) xterm window. The user can switch between different windows and open new ones, but can only see one window at a time See section Multiple screens.

Remote shells and logins

The best way to log onto another system is to use the Secure Shell command `ssh`. This replaces the now obsolete commands:

`rlogin`
: Login onto a remote UNIX system.

`rsh` Open a shell on a remote system (require access rights).

`telnet`
: Open a connection to a remove system using the telnet protocol.

These old commands are insecure andnote very flexible. The Secure Shell offers encryption, strong authentication and greater functionality. It can be used to run a single program on a remote machine, or to login on the remote machine.

```
cube$ ssh metaverse date
cube$ ssh metaverse
```

Text editors

`ed` An ancient line-editor.
`vi` Visual interface to `ed`. This is the only "standard" UNIX text editor supplied by vendors.

`emacs`
: The most powerful UNIX editor. A fully configurable, user programmable editor which works under X11 and on tty-terminals.

`xemacs`
: A pretty version of emacs for X11 windows.

`pico` A tty-terminal only editor, comes as part of the PINE mail package.

`xedit`
: A test X11-only editor supplied with X-windows.

`textedit`
: A simple X11-only editor supplied by Sun Microsystems.

File handling commands

`ls` List files in specified directory (like `dir` on other systems).
`cp` Copy files.
`mv` Move or rename files.
`touch`
: Creates an empty new file if none exists, or updates date and time stamps on existing files.

`rm, unlink`
: Remove a file or link (delete).

`mkdir, rmdir`
: Make or remove a directory. A directory must be empty in order to be able to remove it.

`cat` Concatenate or join together a number of files. The output is written to the standard output by default. Can also be used to simply print a file on screen.

`lp, lpr`
: Line printer. Send a file to the default printer, or the printer defined in the `'PRINTER'` environment variable.

`lpq, lpstat`
: Show the status of the print queue.

File browsing

`more` Shows one screen full at a time. Possibility to search for a string and edit the file. This is like `'type` *`file`* `| more'` in DOS.
`less` An enhanced version of more.
`mc` Midnight commander, a free version of the 'Norton Commander' PC utility for UNIX. (Only for non-serious UNIX users...)
`kfm` A window based file manager with icons and all that nonsense.

Ownership and granting access permission

`chmod`
: Change file access mode.

`chown, chgrp`
: Change owner and group of a file. The GNU version of `chown` allows both these operations to be performed together using the syntax `chown` *`owner.group file`*.

`acl` On newer Unices, Access control lists allow access to be granted on a per-user basis rather than by groups.

Extracting from and rebuilding files

`cut` Extract a column in a table
`paste`

Merge several files so that each file becomes a column in a table.

`sed` A batch text-editor for searching, replacing and selecting text without human intervention.

`awk` A prerunner to the Perl language, for extracting and modifying textfiles.

`rmcr` Strip carriage return (ASCII 13) characters from a file. Useful for converting DOS files to UNIX.

Locating files

`find` Search for files from a specified directory using various criteria.

`locate`
Fast search in a global file database for files containing a search-string.

`whereis`
Look for a command and its documentation on the system.

Disk usage.

`du` Show number of blocks used by a file or files.

`df` Show the state of usage for one or more disk partitions.

Show other users logged on

`users`
Simple list of other users.

`finger`
Show who is logged onto this and other systems.

`who` List of users logged into this system.

`w` Long list of who is logged onto this system and what they are doing.

Contacting other users

`write`
Send a simple message to the named user, end with `CTRL-D`. The command `'mesg n'` switches off messages receipt.

`talk` Interactive two-way conversation with named user.

`irc` Internet relay chat. A conferencing system for realtime multi-user conversations, for addicts and losers.

Mail senders/readers

`mail` The standard (old) mail interface.

`Mail` Another mail interface.

`elm` Electronic Mail program. Lots of functionality but poor support for multimedia.

`pine` Rumours (untrue) are that pine stands for Pine is Not Elm; it actually stands for nothing at all. Improved support for multimedia but very slow and rather stupid at times. Some of the best features of elm have been removed!

`mailtool`
Sun's openwindows client program.

`rmail`

	A mail interface built into the emacs editor.
`netscape mail`	A mail interface built into the netscape navigator.
`zmail`	A commercial mail package.
`tkrat`	A graphical mail reader which supports most MIME types, written in tcl/tk. This program has a nice feel and allows you to create a searchable database of old mail messages, but has a hopeless locking mechanism.

File transfer

`ftp`	The File Transfer program - copies files to/from a remote host.
`ncftp`	An enhanced ftp for anonymous login.

Compilers

`cc`	The C compiler.
`CC`	The C++ compiler.
`gcc`	The GNU C compiler.
`g++`	The GNU C++ compiler.
`javac`	A generator of Java bytecode.
`java`	A Java Virtual Machine.
`ld`	The system linker/loader.
`ar`	Archive library builder.
`dbx`	A symbolic debugger.
`gdb`	The GNU symbolic debugger.
`xxgdb`	The GNU debugger with a window driven front-end.
`ddd`	A motif based front-end to the gdb debugger.

Other interpreted languages

`perl`	Practical extraction an report language.
`tcl`	A perl-like language with special support for building user interfaces and command shells.
`php`	Personal Home Page Tools (officially "PHP: Hypertext Preprocessor"). A server-side HTML-embedded scripting language.
`scheme`	A lisp-like extensible scripting language from GNU.
`mercury`	A prolog-like language for artificial intelligence.

Processes and system statistics

`ps`	List system process table.

`vmstat`
: List kernel virtual-memory statistics.

`netstat`
: List network connections and statistics.

`rpcinfo`
: Show rpc information.

`showmount`
: Show clients mounting local filesystems.

System identity

`uname`
: Display system name and operating system release.

`hostname`
: Show the name of this host.

`domainname`
: Show the name of the local NIS domain. Normally this is chosen to be the same as the BIND/DNS domain, but it need not be.

`nslookup`
: Interrogate the DNS/BIND name service (hostname to IP address conversion).

Internet resources

`archie, xarchie`
: Search the internet ftp database for files.

`xrn, fnews`
: Read news (browser).

`netscape, xmosaic`
: Read world wide web (WWW) (browser).

Text formatting and postscript

`tex, latex`
: Donald Knuth's text formatting language, pronounced "tek" (the x is really a Greek "chi"). Used widely for technical publications. Compiles to dvi (device independent) file format.

`texinfo`
: A hypertext documentation system using tex and "info" format. This is the GNU documentation system. This UNIX guide is written in texinfo!!!

`xdvi` View a tex dvi file on screen.

`dvips`
: Convert dvi format into postscript.

`ghostview, ghostscript`
: View a postscript file on screen.

Picture editors and processors

`xv` Handles, edits and processes pictures in a variety of standard graphics formats (gif, jpg, tiff etc).

Use `xv -quit` to place a picture on your root window.

`xpaint`
: A simple paint program.

`xfig` A line drawing figure editor. Produces postscript, tex, and a variety of other output formats.

`xmgr` A graphing and analysis program.

`xsetroot`
: Load an X-bitmap image into the screen (root window) background. Small images are tiled.

Miscellaneous

`date` Print the date and time.

`ispell`
: Spelling checker.

`xcalc`
: A graphical calculator.

`dc,bc`
: Text-based calculators.

`xclock`
: A clock!

`ping` Send a "sonar" ping to see if another UNIX host is alive.

Terminals

In order to communicate with a user, a shell needs to have access to a terminal. UNIX was designed to work with many different kinds of terminals. Input/output commands in UNIX read and write to a virtual terminal. In *reality* a terminal might be a text-based Teletype terminal (called a *tty* for short) or a graphics based terminal; it might be 80-characters wide or it might be wider or narrower. UNIX take into account these possibility by defining a number of instances of terminals in a more or less object oriented way.

Each user’s terminal has to be configured before cursor based input/output will work correctly. Normally this is done by choosing one of a number of standard terminal types a list which is supplied by the system. In practice the user defines the value of the environment variable `‘TERM’` to an appropriate name. Typical examples are `‘vt100’` and `‘xterm’`. If no standard setup is found, the terminal can always be configured manually using UNIX’s most cryptic and opaque of commands: `‘stty’`.

The job of configuring terminals is much easier now that hardware is more standard. Users’ terminals are usually configured centrally by the system administrator and it is seldom indeed that one ever has to choose anything other than `‘vt100’` or `‘xterm’`.

The X window system

Because UNIX originated before windowing technology was available, the user-interface was not designed with windowing in mind. The X window system attempts to be like a virtual machine park, running a different program in each window. Although the programs appear on one screen, they may in fact be running on UNIX systems anywhere in the world, with only the output being local to the user’s display. The standard shell interface is available by running an X client application called `‘xterm’` which is a graphical front-end to the standard UNIX textual interface.

The `'xterm'` program provides a virtual terminal using the X windows graphical user interface. It works in exactly the same way as a *tty* terminal, except that standard graphical facilities like copy and paste are available. Moreover, the user has the convenience of being able to run a different shell in every window. For example, using the `'rlogin'` command, it is possible to work on the local system in one window, and on another remote system in another window. The X-window environment allows one to cut and paste between windows, regardless of which host the shell runs on.

The components of the X-window system

The X11 system is based on the client-server model. You might wonder why a window system would be based on a model which was introduced for interprocess communication, or network communication. The answer is straightforward.

The designers of the X window system realized that network communication was to be the paradigm of the next generation of computer systems. They wanted to design a system of windows which would enable a user to sit at a terminal in Massachusetts and work on a machine in Tokyo -- and still be able to get high quality windows displayed on their terminal. The aim of X windows from the beginning is to create a *distributed* window environment.

When I log onto my friend's Hewlett Packard workstation to use the text editor (because I don't like the one on my EUNUCHS workstation) I want it to work correctly on my screen, with my keyboard -- even though my workstation is manufactured by a different company. I also want the colours to be right despite the fact that the HP machine uses a completely different video hardware to my machine. When I press the curly brace key `{`, I want to see a curly brace, and not some hieroglyphic because the HP station uses a different keyboard.

These are the problems which X tries to address. In a network environment we need a *common window system* which will work on any kind of hardware, and hide the differences between different machines as far as possible. But it has to be flexible enough to allow us to change all of the things we don't like -- to choose our own colours, and the kind of window borders we want etc. Other windowing systems (like Microsoft windows) ignore these problems and thereby lock the user to a single vendors products and a single operating system. (That, of course, is no accident.)

The way X solves this problem is to use the client server model. Each program which wants to open a window on somebody's compute screen is a client of the *X window service*. To get something drawn on a user's screen, the client asks a server on the host of interest to draw windows for it. No client ever draws anything itself -- it asks the server to do it on its behalf. There are several reasons for this:

- The clients can all talk a common 'window language' or *protocol*. We can hide the difference between different kinds of hardware by making the *machine-specific* part of drawing graphics entirely a problem of implementing the server on the particular hardware. When a new type of hardware comes along, we just need to modify the server -- none of the clients need to be modified.
- We can contact different servers and send our output to different hardware -- thus even though a program is running on a CPU in Tokyo, it can ask the server in Massachusetts to display its window for it.

- When more than one window is on a user’s display, it eventually becomes necessary to move the windows around and then figure out which windows are on top of which other windows etc. If all of the drawing information is kept in a server, it is straightforward to work out this information. If every client drew where it wanted to, it would be impossible to know which window was supposed to be on top of another.

In X, the window manager is a different program to the server which does the drawing of graphics -- but the client-server idea still applies, it just has one more piece to its puzzle.

How to set up X windows

The X windows system is large and complex and not particularly user friendly. When you log in to the system, X reads two files in your home directory which decide which applications will be started what they will look like. The files are called

.Xsession

This file is a shell script which starts up a number of applications as background processes and exits by calling a window manager. Here is a simple example file

```
#!/bin/bash
#
# .xsession file
#
#

PATH="/usr/bin:/bin:/local/gnu/bin:/usr/X11R6/bin"

#
# List applications here, with & at the end
# so they run in the background
#

  xterm -T NewTitle -sl 1000 -geometry 90x45+16+150 -sb &
  xclock &
  xbiff -geometry 80x80+510+0 &
  netscape -iconic&

# Start a window manager. Exec replaces this script with
# the fvwm process, so that it doesn’t exist as a separate
# (useless) process.

  exec /local/bin/fvwm
```

.Xdefaults

This file specifies all of the resources which X programs use. It can be used to change the colours used by applications, or font types etc. The subject of X-resources is a large one and we don’t have time for it here. Here is a simple example, which shows how you can make your over-bright xterm and emacs windows less bright grey shade.

```
xterm*background: LightGrey
Emacs*background: grey92
Xemacs*background: grey92
```

X displays and authority

In the terminology used by X11, every client program has to contact a *display* in order to open a window. A display is a virtual screen which is created by the X server on a particular host. X can create several separate displays on a given host, though most machines only have one.

When an X client program wants to open a window, it looks in the UNIX environment variable `‘DISPLAY’` for the IP address of a host which has an X server it can contact. For example, if we wrote

```
DISPLAY="myhost:0"
export DISPLAY
```

the client would try to contact the X server on ‘myhost’ and ask for a window on display number zero (the usual display). If we wrote

```
DISPLAY="198.112.208.35:0"
export DISPLAY
```

the client would try to open display zero on the X server at the host with the IP address `‘198.112.208.35’`.

Clearly there must be some kind of security mechanism to prevent just anybody from opening windows on someone’s display. X has two such mechanisms:

xhost
: This mechanism is now obsolete. The `‘xhost’` command is used to define a list of hosts which are allowed to open windows on the user’s display. It cannot distinguish between individual users. i.e. the command `xhost yourhost` would allow *anyone* using yourhost to access the local display. This mechanism is only present for backward compatibility with early versions of X windows. Normally one should use the command `xhost -` to exclude all others from accessing the display.

Xauthority
: The Xauthority mechanism has replaced the xhost scheme. It provides a security mechanism which can distinguish individual users, not just hosts. In order for a user to open a window on a display, he/she must have a ticket--called a "magic cookie". This is a binary file called `‘.Xauthority’` which is created in the user’s home directory when he/she first starts the X-windows system. Anyone who does not have a recent copy of this file cannot open windows or read the display of the user’s terminal. This mechanism is based on the idea that the user’s home directory is available via NFS on all hosts he/she will log onto, and thus the owner of the display will always have access to the magic cookie, and will therefore always be able to open windows on the display. Other users must obtain a copy of the file in order to open windows there. The command `xauth` is an interactive utility used for controlling the contents of the `‘.Xauthority’` file. See the `‘xauth’` manual page for more information.

Multiple screens

The window paradigm has been very successful in many ways, but anyone who has used a window system knows that the screen is simply not big enough for all the windows one would like! UNIX has several solutions to this problem.

One solution is to attach several physical screens to a terminal. The X window system can support any number of physical screens of different types. A graphical designer might want a high resolution colour screen for drawing and a black and white screen for writing text, for instance. The disadvantage with this method is the cost of the hardware.

A cheaper solution is to use a window manager such as `‘fwvm’` which creates a virtual screen of unlimited size on a single monitor. As the mouse pointer reaches the edge of the true screen, the window manager replaces the display with a new "blank screen" in which to place windows. A miniaturized image of the windows on a control panel acts as a map which makes it possible to find the applications on the virtual screen.

Yet another possibility is to create virtual displays inside a single window. In other words, one can collapse several shell windows into a single `‘xterm’` window by running the program `‘screen’`. The screen command allows you to start several shells in a single window (using `CTRL-a CTRL-c`) and to switch between them (by typing `CTRL-a CTRL-n`). It is only possible to see one shell window at a time, but it is still possible to cut and paste between windows and one has a considerable saving of space. The `‘screen’` command also allows you to suspend a shell session, log out, log in again later and resume the session precisely where you left off.

Here is a summary of some useful screen commands:

`screen`
: Start the screen server.

`screen -r`
: Resume a previously suspended screen session if possible.

`CTRL-a CTRL-c`
: Start a new shell on top of the others (a fresh ‘screen’) in the current window.

`CTRL-a CTRL-n`
: Switch to the next ‘screen’.

`CTRL-a CTRL-a`
: Switch to the last screen used.

`CTRL-a a`
: When screen is running, `CTRL-a` is used for screen commands and cannot therefore be used in its usual shell meaning of ‘jump to start of line’. `CTRL-a a` replaces this.

`CTRL-a CTRL-d`
: Detach the screen session from the current window so that it can be resumed later. It can be resumed with the `‘screen -r’` command.

`CTRL-a ?`
: Help screen.

Files and access

To prevent all users from being able to access all files on the system, UNIX records information about *who* creates files and also who is allowed to access them later.

Each user has a unique *username* or *loginname* together with a unique *user id* or *uid*. The user id is a number, whereas the login name is a text string -- otherwise the two express the same information. A file belongs to user A if it is *owned* by user A. User A then decides whether or not other users can read, write or execute the file by setting the *protection bits* or the *permission* of the file using the command `chmod`.

In addition to user identities, there are groups of users. The idea of a group is that several named users might want to be able to read and work on a file, without other users being able to access it. Every user is a member of at least one group, called the *login group* and each group has both a textual name and a number (*group id*). The *uid* and *gid* of each user is recorded in the file `/etc/passwd` (See chapter 6). Membership of other groups is recorded in the file `/etc/group` or on some systems `/etc/logingroup`.

Protection bits

The following output is from the command `ls -lag` executed on a SunOS type machine.

```
lrwxrwxrwx  1 root     wheel          7 Jun  1  1993 bin -> usr/bin
-r--r--r--  1 root     bin       103512 Jun  1  1993 boot
drwxr-sr-x  2 bin      staff      11264 May 11 17:00 dev
drwxr-sr-x 10 bin      staff       2560 Jul  8 02:06 etc
drwxr-sr-x  8 root     wheel        512 Jun  1  1993 export
drwx------  2 root     daemon       512 Sep 26  1993 home
-rwxr-xr-x  1 root     wheel     249079 Jun  1  1993 kadb
lrwxrwxrwx  1 root     wheel          7 Jun  1  1993 lib -> usr/lib
drwxr-xr-x  2 root     wheel       8192 Jun  1  1993 lost+found
drwxr-sr-x  2 bin      staff        512 Jul 23  1992 mnt
dr-xr-xr-x  1 root     wheel        512 May 11 17:00 net
drwxr-sr-x  2 root     wheel        512 Jun  1  1993 pcfs
drwxr-sr-x  2 bin      staff        512 Jun  1  1993 sbin
lrwxrwxrwx  1 root     wheel         13 Jun  1  1993 sys->kvm/sys
drwxrwxrwx  6 root     wheel        732 Jul  8 19:23 tmp
drwxr-xr-x 27 root     wheel       1024 Jun 14  1993 usr
drwxr-sr-x 10 bin      staff        512 Jul 23  1992 var
-rwxr-xr-x  1 root     daemon   2182656 Jun  4  1993 vmUNIX
```

The first column is a textual representation of the protection bits for each file. Column two is the number of hard links to the file (See exercises below). The third and fourth columns are the user name and group name and the remainder show the file size in bytes and the creation date. Notice that the directories `/bin` and `/sys` are symbolic links to other directories.

There are sixteen protection bits for a UNIX file, but only twelve of them can be changed by users. These twelve are split into four groups of three. Each three-bit number corresponds to one *octal* number.

The leading four invisible bits gives information about the type of file: is the file a *plain file*, a *directory* or a *link*. In the output from `ls` this is represented by a single character: `-`, `d` or `l`.

The next three bits set the so-called *s-bits* and *t-bit* which are explained below.

The remaining three groups of three bits set flags which indicate whether a file can be read `'r'`, written to `'w'` or executed `'x'` by (i) the user who created them, (ii) the other users who are in the group the file is marked with, and (iii) any user at all.

For example, the permission

```
Type Owner Group Anyone
  d  rwx   r-x   ---
```

tells us that the file is a directory, which can be read and written to by the owner, can be read by others in its group, but not by anyone else.

Note about directories. It is impossible to `cd` *to a directory unless the* `x` *bit is set. That is, directories must be 'executable' in order to be accessible.*

Here are some examples of the relationship between binary, octal and the textual representation of file modes.

```
Binary  Octal   Text

 001      1       x
 010      2       w
 100      4       r
 110      6      rw-
 101      5      r-x
  -      644   rw-r--r--
```

It is well worth becoming familiar with the octal number representation of these permissions.

chmod

The `chmod` command changes the permission or *mode* of a file. Only the owner of the file or the superuser can change the permission. Here are some examples of its use. Try them.

```
# make read/write-able for everyone
chmod a+w myfile

# add the 'execute' flag for directory
chmod u+x mydir/

# open all files for everyone
chmod 755 *

# set the s-bit on my-dir's group
chmod g+s mydir/

# descend recursively into directory opening all files
chmod -R a+r dir
```

Umask

When a new file gets created, the operating system must decide what default protection bits to set on that file. The variable `umask` decides this. `umask` is normally set by each user in his or her `.cshrc` file (see next chapter). For example

```
umask 077    # safe
umask 022    # liberal
```

According the UNIX documentation, the value of `umask` is `'XOR'`ed (exclusive `'OR'`) with a value of `666 & umask` for plain files or `777 & umask` for directories in order to find out the standard protection. Actually this is not quite true: `'umask'` only removes bits, it never sets bits which were not already set in `666`. For instance

umask	Permission
077	600 (plain)
077	700 (dir)
022	644 (plain)
022	755 (dir)

The correct rule for computing permissions is not XOR but `'NOT AND'`.

Making programs executable

A UNIX program is normally executed by typing its pathname. If the `x` execute bit is not set on the file, this will generate a 'Permission denied' error. This protects the system from interpreting nonsense files as programs. To make a program executable for someone, you must therefore ensure that they can execute the file, using a command like

```
chmod u+x filename
```

This command would set execute permissions for the owner of the file;

```
chmod ug+x filename
```

would set execute permissions for the owner and for any users in the same group as the file. Note that script programs must also be readable in order to be executable, since the shell has the interpret them by reading.

chown and chgrp

These two commands change the ownership and the group ownership of a file. Only the superuser can change the ownership of a file on most systems. This is to prevent users from being able to defeat quota mechanisms. (On some systems, which do not implement quotas, ordinary users can give a file away to another user but not get it back again.) The same applies to group ownership.

Making a group

Normally users other than root cannot define their own groups. This is a weakness in UNIX from older times which no one seems to be in a hurry to change.

s-bit and t-bit (sticky bit)

The `s` and `t` bits have special uses. They are described as follows.

Octal	Text	Name

```
4000      chmod u+s  Setuid bit
2000      chmod g+s  Setgid bit
1000      chmod +t   Sticky bit
```

The effect of these bits differs for plain files and directories and differ between different versions of UNIX. You should check the manual page `man sticky` to find out about your system! The following is common behaviour.

For executable files, the setuid bit tells UNIX that *regardless of who runs the program* it should be executed with the permissions and rights of owner of the file. This is often used to allow normal users limited access to `root` privileges. A *setuid-root* program is executed as `root` for any user. The setgid bit sets the group execution rights of the program in a similar way.

In BSD UNIX, if the setgid bit is set on a directory then any new files created in that directory assume the group ownership of the parent directory and not the logingroup of the user who created the file. This is standard policy under system 5.

A directory for which the sticky bit is set restrict the deletion of files within it. A file or directory inside a directory with the t-bit set can only be deleted or renamed by its owner or the superuser. This is useful for directories like the mail spool area and `/tmp` which must be writable to everyone, but should not allow a user to delete another user's files.

(Ultrix) If an executable file is marked with a sticky bit, it is held in the memory or system swap area. It does not have to be fetched from disk each time it is executed. This saves time for frequently used programs like `ls`.

(Solaris 1) If a non-executable file is marked with the sticky bit, it will *not* be held in the disk page cache -- that is, it is never copied from the disk and held in RAM but is written to directly. This is used to prevent certain files from using up valuable memory.

On some systems (e.g. ULTRIX), only the superuser can set the sticky bit. On others (e.g. SunOS) any user can create a sticky directory.

Bourne Again shell

The Bourne Again shell (Bash) is the command interpreter which you use to run programs and utilities. It contains a simple programming language for writing tailor-made commands, and allows you to join together UNIX commands with pipes. It is a configurable environment, and once you know it well, it is the most efficient way of working with UNIX.

The Bourne Again shell was written by the Free Software Foundation as a part of the GNU project and Bash is the default shell in most GNU/Linux distributions. Because of its command line editing features, it is much more efficient for interactive use than Bourne shell, the original UNIX shell. Most of the system scripts in UNIX are written in the Bourne shell. Although Bash includes many extensions and features not found in the Bourne shell, it maintains compatibility with it so that you can run Bourne shell scripts under Bash. On many GNU/Linux systems Bourne shell (`‘/bin/sh’`) is symbolically linked to Bash (`‘/bin/bash’`) so that the scripts that require the presence of the Bourne shell still run. If you want to write a platform independent shell script able to run on as many UNIX variants as possible, you should stick to Bourne shell syntax and avoid the Bash extensions.

'~/.bashrc' and '~/.bash_profile' files

When you log on to a GNU/Linux system and your login shell is defined in '/etc/passwd' to be Bash, it first executes commands in the '/etc/profile' file. It then searches for the '~/.bash_profile,' '~/.bash_login' or '~/.profile' file, in this order, and executes commands in the first of these that is found and is readable. When a login exits, it executes commands in the '~/.bash_logout' file.

When you start an non-login interactive Bash shell, it only executes commands in the '~/.bashrc' file, if it exists and is readable. However, this shell inherits any environment (exported) variables from the parent shell, so environment variables set in '/etc/profile' and '~/.bash_profile' are passed onto the non-login shells and later to its subshells.

Here is a very simple example '~/.bashrc' file:

```
#
# .bashrc - read in by every bash that starts.
#

umask 077             # Set the default file creation mask
PATH="~/bin:$PATH"    # Inserts own bin directory first in PATH


PS1="‘uname‘:\h\$ "  # prompt
PS2="\h > "          # prompt for foreach and while
PRINTER=myprinter

# Aliases are shortcuts to UNIX commands
alias h=history
alias ll="ls -l"
alias cp='cp -i'
alias rm='rm -i'
alias c='ssh cube'
```

In order to make sure your '~/.bashrc' file is read when logging on with ssh to another machine, you may start your '~/.bash_profile' file like this:

```
#
# .bash_profile - read in every login.
#

if [ -f ~/.bashrc ]
then
   source ~/.bashrc  # runs .bashrc as if they where
                     # typed into this file
fi
```

Variables and export

Shell variables are defined using the syntax

```
VARIABLE="username is"
```

```
myname="‘whoami‘"
```

It is important that there be no space between the variable and the equals sign. These variables are then referred to using the dollar ‘$’ symbol.

```
$ echo "My $VARIABLE $myname"
My username is mark
```

When assigning values to variables the dollar symbol is never used. By default these variables are *local* - that is they will not be passed on to programs and sub-shells running under the current shell. To make them global (so that child processes will inherit them) we use the command

```
export VARIABLE
```

This adds the variable to the process *environment*. Under Bash (but not under the old Bourne shell) it is also possible to declare a variable to be global on a single line by

```
export GLOBALVAR="global"
```

The command

```
set -a
```

changes the default so that all variables, after the command are created *global*.

Arrays or lists are often simulated in Bourne shell by sandwiching the colon ‘:’ symbol between items

```
PATH=/bin:/usr/bin:/etc:/local/bin:.

LD_LIBARAY_PATH=/usr/lib:/usr/openwin/lib:/local/lib
```

but there is no real facility for arrays in the Bourne shell. Note that the UNIX ‘cut’ command can be used to extract the elements of the list. Loops can also read such lists directly See section Loops in Bash. However, Bash version 2.x supports arrays as seen in the next section.

The value of a variable is given by the dollar symbol. It is also possible to use *curly braces* around the variable name to ‘protect’ the variable from interfering text. For example:

```
$ animal=worm
$ echo book$animal
bookworm
$ thing=book
$ echo $thingworm
                       (nothing..)
$ echo ${thing}worm
bookworm
```

Default values can be given to variables in the Bourne shell. The following commands illustrate this.

```
echo ${var-"No value set"}
echo ${var="Octopus"}
echo ${var+"Forced value"}
echo ${var?"No such variable"}
```

The first of these prints out the contents of `‘$var’`, if it is defined. If it is not defined the variable is substituted for the string "No value set". The value of `‘var’` is not changed by this operation. It is only for convenience.

The second command has the same effect as the first, but here the value of `‘$var’` is actually changed to "Octopus" if `‘$var’` is not set.

The third version is slightly peculiar. If `‘$var’` is *already* set, its value will be forced to be "Forced value", otherwise it is left undefined.

Finally the last instance issues an error message "No such variable" if `‘$var’` is not defined.

In Bash 2.x it is possible to extract parts of the string a variable is set to using the construction `${variable:offset:length}|` as shown in the next example.

```
var="abcdefg"
middle=${var:2:3}
echo $middle
cde
```

An offset of 2 skips the first 2 characters and a string of length 3 is extracted from the middle of the string.

Bash arrays

The original Bourne shell does not have arrays. Bash version 2.x does have arrays, however. An array can be assigned from a string of words separated by whitespaces or the individual elements of the array can be set individually.

```
colours=(red white green)
colours[3]="yellow"
```

An element of the array must be referred to using curly braces.

```
echo ${colours[1]}
white
```

Note that the first element of the array has index 0. The set of all elements is referred to by `${colours[*]}`.

```
echo ${colours[*]}
red white green yellow
echo ${#colours[*]}
4
```

As seen the number of elements in an array is given by `${#colours[*]}`.

Stdin, stdout, stderr and redirection to and from files

When the shell starts up, it inherits three files: `‘stdin’`, `‘stdout’`, and `‘stderr’`. Standard input normally comes from the keyboard. Standard output and standard error normally go to the screen. There are times you want to read input from a file or send output of errors to a file. This can be accomplished by using *I/O redirection*.

In Bash and the Bourne shell, the standard input/output files are referred to by numbers rather than by names.

stdin
: File number 0

stdout
: File number 1

stderr
: File number 2

The default routes for these files can be changed by redirection. The output of the command `echo` is by default sent to the screen, that is the stdout with file number 1 is sent to the screen. Using redirection operators it is possible to redirect the standard out of `echo` to where we want it. We can send output to a file with the following command.

```
echo "should be sent to a file" > file.txt
```

This creates a new file `‘file.txt’` containing the string ’should be sent to a file’. The redirection operator could have been given as `1>`, but it is understood that standard out is meant when skipping the number of the file handle. The single ’>’ always creates a new file, while ’>>’ appends to the end of a file.

If you had mistyped the command `echo` the result would have been:

```
ehco "should be sent to a file" > file.txt
bash: ehco: command not found
```

The standard error with file handle 2 is by default sent to the screen, independent of where standard out (1) is sent. If you like you can redirect stdout to another or the same file.

```
ehco "should be sent to a file" > file.txt 2> error.txt
cat error.txt
bash: ehco: command not found
```

There are several ways to send stderr to the same file as stdin is redirected to. The following three commands are equivalent.

```
ehco "should be sent to a file" >& file.txt
ehco "should be sent to a file" >  file.txt 2> file.txt
ehco "should be sent to a file" >  file.txt 2>&1
```

The string `2>&1` means that stderr(2) should be sent to the same file as stdout(1). This is the only why to do this under the Bourne shell and this construction is therefore often seen in system shell scripts.

Furthermore it is possible to force a command which by default takes standard input from the keyboard, to read input from a file by redirecting stdin. The mail-command expects input from keyboard, but the ’<’ redirection operator makes it send the password file to the user mark:

```
/bin/mail mark < /etc/passwd
```

The following table summarizes the most important redirection operators:

Redirection operator	What it does
`<`	Redirects input
`>`	Redirects output
`>>`	Appends output
`2>`	Redirects error
`>&`	Redirects output and error (Bash only)
`2>&1`	Redirects error where output (1) is going

Pipes

A *pipe* takes the output from the command on the left-hand side of the pipe symbol and sends it to the input of the command on the right-hand side of the pipe symbol. A pipeline can consist of several pipes and this makes pipes a very powerful tool. It enables us to combine all the small and efficient UNIX commands in any thinkable way. If you want to count the number of people logged on, you could save the output of the command `who` in the temporary file `'tmp'`, use `wc -l` to count the number of lines in `'tmp'` and finally remove the temporary file.

```
$ who > tmp
$ wc -l tmp
      4 tmp
$ rm tmp
```

Using a pipe saves disk space and time: the stdout from `who` can be redirected to the stdin of `wc -l` through a pipe and there is no need for temporarily storing the output from `who`.

```
$ who | wc -l
      4
```

Most UNIX-commands are constructed with piping in mind and this makes it possible to solve complex tasks easily, by joining commands along a pipeline. Consider the following pipeline:

```
cat big.jpg | djpeg | pnmscale -pixels 150000 | cjpeg > small.jpg
```

The command `cat` sends the large JPEG-image to `djpeg` which decompresses it and sends the resulting bitmap to stdout. The stream of data floats through the next pipe to `pnmscale` which scales the bitmap image down to the given size. The scaled image is piped to the command `cjpeg` which compresses the standard input and finally produces a JPEG-image of reduced size which is stored in the file `'small.jpg'`.

Command history

The history feature in Bash means that you do not have to type commands over and over again. You can use the `UP ARROW` key to browse back through the list of commands you have typed previously and the keys `LEFT ARROW` and `RIGHT ARROW` to edit these commands.

In addition there are a couple of commands which selects commands from the history list.

`‘!!’` Execute the last command again.
`‘!4’` Execute command number 4.

The first of these simply repeats the last command. The second command gives an absolute number. The absolute command number can be seen by typing `‘history’`.

Command/filename completion

In Bash you can save hours worth of typing errors by using the completion mechanism. This feature is based on the `TAB` key.

The idea is that if you type half a filename and press `TAB`, the shell will try to guess the remainder of the filename. It does this by looking at the files which match what you have already typed and trying to fill in the rest. If there are several files which match, the shell sounds the "bell" or beeps. You can then type `TAB` twice to obtain a list of the possible alternatives. Here is an example: suppose you have just a single file in the current directory called `‘very_long_filename’`, typing

```
more TAB
```

results in the following appearing on the command line

```
more very_long_filename
```

The shell was able to identify a unique file. Now suppose that you have two files called `‘very_long_filename’` and `‘very_big_filename’`, typing

```
more TAB
```

results in the following appearing on the command line

```
more very_
```

and the shell beeps, indicating that the choice was not unique and a decision is required. Next, you type `TAB` twice(3) to see which files you have to choose from and the shell lists them and returns you to the command line, exactly where you were. You now choose `‘very_long_filename’` by typing `‘l’`. This is enough to uniquely identify the file. Pressing the `TAB` key again results in

```
more very_long_filename
```

on the screen. As long as you have written enough to select a file uniquely, the shell will be able to complete the name for you.

Completion also works on shell commands, but it is a little slower since the shell must search through all the directories in the command path to complete commands.

Single and double quotes

Two kinds of quotes can be used in shell apart from the backward quotes we mentioned above. The essential difference between them is that certain shell commands work inside *double* quotes but not inside single quotes. For example

```
cube$ echo /etc/rc*
/etc/rc.boot /etc/rc0.d /etc/rc1.d /etc/rc2.d /etc/rc3.d /etc/rc4.d

cube$ echo "/etc/rc*"
/etc/rc*

cube$ echo "‘whoami‘  -- my name is $USER"
mark  -- my name is mark

cube$ echo ’‘whoami‘  -- my name is $USER’
‘whoami‘  -- my name is $USER
```

We see that the single quotes prevent *variable substitution* and *sub-shells*. Wildcards do not work inside either single or double quotes.

Job control, break key, `‘fg’`, `‘bg’`

So far we haven’t mentioned UNIX’s ability to multitask. In the Bourne shell (`‘sh’`) there are no facilities for controlling several user processes. Bash provides some commands for starting and stopping processes. These originate from the days before windows and X11, so some of them may seem a little old-fashioned. They are still very useful nonetheless.

Let’s begin by looking at the commands which are true for any shell. Most programs are run in the *foreground* or *interactively*. That means that they are connected to the standard input and send their output to the standard output. A program can be made to run in the background, if it does not need to use the standard I/O. For example, a program which generates output and sends it to a file could run in the background. In a window environment, programs which create their own windows can also be started as background processes, leaving standard I/O in the shell free.

Background processes run independently of what you are doing in the foreground.

UNIX Processes and BSD signals

A background process is started using the special character `‘&’` at the end of the command line.

```
find / -name ’*lib*’ -print >& output  &
```

The final `‘&’` on the end of this line means that the job will be run in the background. Note that this is not confused with the redirection operator `‘>&’` since it must be the last character on the line. The command above looks for any files in the system containing the string ‘lib’ and writes the list of files to a file called ‘output’. This might be a useful way of searching for missing libraries which you want to include in your environment variable `‘LD_LIBRARY_PATH’`. Searching the entire disk from the root directory `‘/’` could take a long time, so it pays to run this in the background.

If we want to see what processes are running, we can use the `‘ps’` command. `‘ps’` without any arguments lists all of your processes, i.e. all processes owned by the user name you logged in with in the current shell. `'ps'` takes many options, for instance `'ps auxg'` will list all processes in gruesome detail (The "g" is for group, not gruesome!). `'ps'` reads the kernel's process tables directly.

Processes can be stopped and started, or killed one and for all. The `'kill'` command does this. There are, in fact, two versions of the `'kill'` command. One of them is built into Bash and the other is not. If you use Bash then you will never care about the difference. We shall nonetheless mention the special features of Bash built-ins below. The kill command takes a number called a *signal* as an argument and another number called the *process identifier* or *PID* for short. Kill send signals to processes. Some of these are fatal and some are for information only. The two commands

```
kill -15 127
kill 127
```

are identical. They both send signal 15 to PID 127. This is the normal *termination* signal and it is often enough to stop any process from running.

Programs can choose to ignore certain signals by trapping signals with a special handler. One signal they cannot ignore is signal 9.

```
kill -9  127
```

is a sure way of killing PID 127. Even though the process dies, it may not be removed from the kernel's process table if it has a parent (see next section).

Here is the complete list of signals which the Linux kernel send to processes in different circumstances.

```
#define SIGHUP      1       /* Hangup (POSIX). */
#define SIGINT      2       /* Interrupt (ANSI). */
#define SIGQUIT     3       /* Quit (POSIX). */
#define SIGILL      4       /* Illegal instruction (ANSI). */
#define SIGTRAP     5       /* Trace trap (POSIX). */
#define SIGABRT     6       /* Abort (ANSI). */
#define SIGIOT      6       /* IOT trap (4.2 BSD). */
#define SIGBUS      7       /* BUS error (4.2 BSD). */
#define SIGFPE      8       /* Floating-point exception (ANSI). */
#define SIGKILL     9       /* Kill, unblockable (POSIX). */
#define SIGUSR1     10      /* User-defined signal 1 (POSIX). */
#define SIGSEGV     11      /* Segmentation violation (ANSI). */
#define SIGUSR2     12      /* User-defined signal 2 (POSIX). */
#define SIGPIPE     13      /* Broken pipe (POSIX). */
#define SIGALRM     14      /* Alarm clock (POSIX). */
#define SIGTERM     15      /* Termination (ANSI). */
#define SIGSTKFLT   16      /* Stack fault. */
#define SIGCLD      SIGCHLD /* Same as SIGCHLD (System V). */
#define SIGCHLD     17      /* Child status has changed (POSIX). */
#define SIGCONT     18      /* Continue (POSIX). */
#define SIGSTOP     19      /* Stop, unblockable (POSIX). */
#define SIGTSTP     20      /* Keyboard stop (POSIX). */
#define SIGTTIN     21      /* Background read from tty (POSIX). */
#define SIGTTOU     22      /* Background write to tty (POSIX). */
#define SIGURG      23      /* Urgent condition on socket (4.2 BSD). */
#define SIGXCPU     24      /* CPU limit exceeded (4.2 BSD). */
#define SIGXFSZ     25      /* File size limit exceeded (4.2 BSD). */
#define SIGVTALRM   26      /* Virtual alarm clock (4.2 BSD). */
#define SIGPROF     27      /* Profiling alarm clock (4.2 BSD). */
```

```
#define SIGWINCH   28      /* Window size change (4.3 BSD, Sun). */
#define SIGPOLL    SIGIO   /* Pollable event occurred (System V). */
#define SIGIO      29      /* I/O now possible (4.2 BSD). */
#define SIGPWR     30      /* Power failure restart (System V). */
#define SIGSYS     31      /* Bad system call. */
```

We have already mentioned 15 and 9 which are the main signals for users. Signal 1, or `‘HUP’` can be sent to certain programs by the superuser. For instance

```
kill -1   <inetd>
kill -HUP <inetd>
```

which forces `‘inetd’` to reread its configuration file. Sometimes it is useful to *suspend* a process temporarily and then *restart* it later.

```
kill -20 <PID>       # suspend process <PID>
kill -18 <PID>       # resume process <PID>
```

Child Processes and zombies

When you start a process from a shell, regardless of whether it is a background process or a foreground process, the new process becomes a *child* of the original shell. Remember that the shell is just a UNIX process itself. Moreover, if one of the children starts a new process then it will be a child of the child (a grandchild?)! Processes therefore form *hierarchies*. Several children can have a common *parent*.

If we kill a parent, then (unless the child has detached itself from the parent) all of its children die too. If a child dies, the parent is not affected. Sometimes when a child is killed, it does not die but becomes "defunct" or a zombie process. This means that the child has a parent which is *waiting* for it to finish. If the parent has not yet been informed that the child has died, for example because it has been suspended itself, then the dead child is not removed from the kernel’s process table. When the parent wakes up and receives the message that the child has terminated, the process entry for the dead child can be removed.

Bash builtins: `‘jobs’`, `‘kill’`, `‘fg’`,`‘bg’`, break key

Now let’s look at some commands which are built into Bash for starting and stopping processes. Bash refers to user programs as ‘jobs’ rather than processes -- but there is no real difference. The added bonus of Bash is that each shell has a *job number* in addition to its PID. The job numbers are simpler and are private for the shell, whereas the PIDs are assigned by the kernel and are often very large numbers which are difficult to to remember. When a command is executed in the shell, it is assigned a job number. If you never run any background jobs then there is only ever one job number: 1, since every job exits before the next one starts. However, if you run background tasks, then you can have several jobs "active" at any time. Moreover, by *suspending* jobs, Bash allows you to have several interactive programs running on the same terminal -- the `‘fg’` and `‘bg’` commands allow you to move commands from the background to the foreground and vice-versa.

Take a look at the following shell session.

```
cube$ emacs myfile&
[3] 771
cube$
```

```
  ( other commands ... , edit myfile and close emacs )
```

When a background job is done, the shell prints a message at a suitable moment between prompts.

```
[3]+  Done                    emacs myfile
cube$
```

This tells you that job number 1 finished normally. If the job exits abnormally then the word 'Done' may be replaced by some other message. For instance, if you kill the job, it will say

```
cube$ kill %3
cube$
[3]+  Terminated              emacs myfile
cube$
```

You can list the jobs you have running using the `'jobs'` command. The output looks something like

```
cube$ jobs
[1]   Terminated              xdvi unix
[2]   Running                 xemacs unix.texinfo &
[3]   Running                 xterm -sb -sl 10000 &
[4]   Running                 ghostview &
[5]   Running                 netscape &
[6]   Running                 xterm -sb -sl 10000 &
[7]   Running                 xemacs fil &
[8]+  Stopped                 emacs unix.log
[9]-  Running                 gimp &
```

To suspend a program which you are running in the foreground you can type `CTRL-z` (this is like sending a `'kill -20'` signal from the keyboard). (4) You can suspend any number of programs and then restart them one at a time using `'fg'` and `'bg'`. If you want job 5 to be restarted in the foreground, you would type

```
fg %5
```

When you have had enough of job 5, you can type CTRL-z to suspend it and then type

```
fg %6
```

to activate job 6. Provided a job does not want to send output to `'stdout'`, you can restart any job in the background, using a command like.

```
bg %4
```

This method of working was useful before windows were available. Using `'fg'` and `'bg'`, you can edit several files or work on several programs without have to quit to move from one to another.

See also some related commands for batch processing `'at'`, `'batch'` and `'atq'`, `'cron'`.

NOTE: `CTRL-c` sends a `'kill -2'` signal, which send a standard interrupt message to a program. This is always a safe way to interrupt a shell command.

Arithmetic in Bash

In Bourne shell arithmetic is performed entirely ‘by proxy’. To evaluate an expression we call the `‘expr’` command or the `‘bc’` precision calculator. Here are some examples of `‘expr’`

```
a=‘expr $a+1‘                 # increment a
a=‘expr 4 + 10 \* 5‘          # 4+10*5
check = ‘expr $a \> $b‘       # true=1, false=0. True if $a > $b
```

`‘expr’` is very sensitive to spaces and backslash characters and this makes it a bit awkward to do arithmetic under the Bourne shell.

Bash 2.0 provides a new and simpler way to do arithmetic using double parentheses. If you surround any integer arithmetic expression as in `(( x = y + 1 ))`, you can perform most arithmetic operations with the same syntax as in Java and C.

```
(( x = 1 ))
echo $x
1
(( x++ ))
(( y = 4*x ))
echo $y
8
```

Note that you do not need to use the dollar symbol to refer to a variable within the double parentheses (but you may do it) and that spaces are allowed.

```
(( sum = 2 ))
(( total = 4*$sum + sum ))
echo $total
10
```

The variables within double parentheses are throughout treated as integers. Assigning a float value like 2.5 to a variable results in an syntax error while assigning a string to a variable cause the string to be stored as zero.

Scripts and arguments

Scripts are created by making an executable file which begins with the sequence of characters

```
#!/bin/bash
```

This construction is quite general: any executable file which begins with a sequence

```
#!myprogram -option
```

will cause the shell to attempt to execute

```
myprogam -option filename
```

where *filename* is the name of the file.

If a script is to accept arguments then these can be referred to as `‘ $1 $2 $3..$9’`. There is a logical limit of nine arguments to a Bourne script, but Bash handles the next arguments as `‘${10}’`. `‘$0’` is the name of the script itself.

Here is a simple Bash script which prints out all its arguments.

```
#!/bin/bash
#
# Print all arguments (version 1)
#

for arg in $*
do
  echo Argument $arg
done

echo Total number of arguments was $#
```

The `‘$*’` symbol stands for the entire list of arguments and `‘$#’` is the total number of arguments.

Another way of achieving the same is to use the `‘shift’` command. We shall meet this again in the Perl programming language. `‘shift’` takes the first argument from the argument list and deletes it, moving all of the other arguments down one number -- this is how we can handle long lists of arguments in the Bourne shell.

```
#!/bin/bash
#
#  Print all arguments (version 2)
#

while ( true )
do
  arg=$1;
  shift;
  echo $arg was an argument;
  if [ $# -eq 0 ]; then
    break
  fi
done
```

Return codes

All programs which execute in UNIX return a value through the C `‘return’` command. There is a convention that a return value of zero (0) means that everything went well, whereas any other value implies that some error occurred. The return value is usually the value returned in `‘errno’`, the external error variable in C.

Shell scripts can test for these values either by placing the command directly inside an ‘if’ test, or by testing the variable `‘$?’` which is always set to the return code of the last command. Some examples are given following the next two sections.

Tests and conditionals

Bash and the Bourne shell has an array of tests. They are written as follows. Notice that `'test'` is itself not a part of the shell, but is a program which works out conditions and provides a return code. See the manual page on `'test'` for more details.

`test -f` *`file`*
: True if the file is a plain file

`test -d` *`file`*
: True if the file is a directory

`test -r` *`file`*
: True if the file is readable

`test -w` *`file`*
: True if the file is writable

`test -x` *`file`*
: True if the file is executable

`test -s` *`file`*
: True if the file contains something

`test -g` *`file`*
: True if setgid bit is set

`test -u` *`file`*
: True if setuid bit is set

`test s1 = s2`
: True if strings s1 and s2 are equal

`test s1 != s2`
: True if strings s1 and s2 are unequal

`test` *`x`* `-eq` *`y`*
: True if the integers *x* and *y* are numerically equal

`test` *`x`* `-ne` *`y`*
: True if integers are not equal

`test` *`x`* `-gt` *`y`*
: True if *x* is greater than *y*

`test` *`x`* `-lt` *`y`*
: True if *x* is less than *y*

`test` *`x`* `-ge` *`y`*
: True if *x>=y*

`test` *`x`* `-le` *`y`*
: True if *x* <= *y*

`!`	Logical NOT operator
`-a`	Logical AND
`-o`	Logical OR

Note that an alternate syntax for writing these commands if to use the square brackets, instead of writing the word test.

```
 [ $x -lt $y ]   "=="    test $x -lt $y
```

Just as with the arithmetic expressions, Bash 2.x provides a syntax for conditionals which are more similar to Java and C. While arithmetic C-like expressions can be used within double parentheses, C-like tests can be used within double square brackets.

```
 [[ $var == "OK" || $var == "yes" ]]
```

This C-like syntax is not allowed in the Bourne shell, but is equivalent to

```
[ $var = "OK" -o $var = "yes" ]
```

which is valid in both shells.

Arithmetic C-like tests can be used within double parentheses so that under Bash 2.x the following tests are equivalent:

```
 [ $x -lt $y ]   "==" (( x < y ))
```

Conditional structures

The conditional structures have the following syntax.

```
if UNIX-command
then
   command
else
   commands
fi
```

The `‘else’` clause is, of course, optional. As noted before, the first UNIX command could be *anything*, since every command has a return code. The result is TRUE if it evaluates to *zero* and false otherwise (in contrast to the conventions in most languages). Multiple tests can be made using

```
if UNIX-command
then
   commands
elif UNIX-command
then
   commands
elif UNIX-command
then
   commands
else
   commands
fi
```

where `‘elif’` means ‘else-if’.

The equivalent of the C-school’s `‘switch’` statement is a more Pascal-like `‘case’` structure.

```
case UNIX-command-or-variable in

   wildcard1) commands ;;
   wildcard2) commands ;;
   wildcard3) commands ;;

esac
```

This structure uses the wildcards to match the output of the command or variable in the first line. The first pattern which matches gets executed.

Input from the user in Bash

In shell you can read the value of a variable using the `‘read’` command, with syntax

```
read variable
```

This reads in a string from the keyboard and terminates on a newline character. Under the old Bourne shell another way to do this is to use the `‘input’` command to access a particular logical device. The keyboard device in the current terminal is `‘/dev/tty’`, so that one writes

```
variable = ‘line < /dev/tty‘
```

which fetches a single line from the user. The command `line` is however not available in most GNU/Linux distributions.

Here are some examples of these commands. First a program which asks yes or no...

```
#!/bin/bash
#
# Yes or no
#

echo "Please answer yes or no: "

read answer

case $answer in

  y* | Y* | j* | J* )  echo YES!! ;;

  n* | N* )            echo NO!! ;;

  *)                   echo "Can’t you answer a simple question?"

esac

echo The end
```

Notice the use of pattern matching and the `‘|’` ‘OR’ symbol.

```
#!/bin/bash
#
# Kernel check
#

if test ! -f /vmUNIX          # Check that the kernel is there!
then
   echo "This is not BSD UNIX...hmmm"
   if [ -f /hp-ux ]
   then
      echo "It’s a Hewlett Packard machine!"
   fi
elif [ -w /vmUNIX ]
then
   echo "HEY!! The kernel is writable my me!";
else
```

```
   echo "The kernel is write protected."
   echo "The system is safe from me today."
fi
```

Loops in Bash

The loop structures in Bash and in the Bourne shell have the following syntax.

```
while UNIX-command
do
  commands
done
```

The first command will most likely be a test but, as before, it could in principle be any UNIX command. The `‘until’` loop, reminiscent of BCPL, carries out a task until its argument evaluates to TRUE.

```
until UNIX-command
do
    commands
done
```

Finally the `‘for’` structure has already been used above.

```
for variable in list
do
   commands
done
```

Often we want to be able to use an array of values as the list which `for` parses, but Bourne shell has no array variables. This problem is usually solved by making a long string separated by, for example, colons. For example, the `$PATH` variable has the form

```
PATH = /usr/bin:/bin:/local/gnu/bin
```

Bourne shell allows us to split such a string on whatever character we wish. Normally the split is made on spaces, but the variable `‘IFS’` can be defined with a replacement. To make a loop over all directories in the command path we would therefore write

```
IFS=:

for name in $PATH; do

   commands

done
```

The best way to gain experience with these commands is through some examples.

```
#!/bin/bash
#
# Get text from user repeatedly
#

echo "Type away..."
```

```
while read TEXT
do

   echo You typed $TEXT

   if [ "$TEXT" = "quit" ]; then
      echo "(So I quit!)"
      exit 0
   fi

done

echo "HELP!"
```

This very simple script is a typical use for a while-loop. It gets text repeatedly until the user type ‘quit’. Since read never returns ‘false’ unless an error occurs or it detects an EOF (end of file) character `CTRL-D`, it will never exit without some help from an ‘if’ test. If it does receive a `CTRL-D` signal, the script prints ‘HELP!’.

```
#!/bin/bash
#
# Watch in the background for a particular user
# and give alarm if he/she logs in
#
# To be run in the background, using &
#

if [ $# -ne 1 ]; then
  echo "Give the name of the user as an argument" > /dev/tty
  exit 1
fi

echo "Looking for $1"

until users | grep -s $1
do
   sleep 60
done

echo "!!! WAKE UP !!!" > /dev/tty
echo "User $1 just logged in" > /dev/tty
```

This script uses `‘grep’` in ‘silent mode’ (-s option). i.e. grep never writes anything to the terminal. The only thing we are interested in is the return code the piped command produces. If ‘grep’ detects a line containing the username we are interested in, then the result evaluates to TRUE and the sleep-loop exits.

Our final example is the kind of script which is useful for a system administrator. It transfers over the Network Information Service database files so that a slave server is up to date. All we have to do is make a list of the files and place it in a `‘for’` loop. The names used below are the actual names of the NIS maps, well known to system administrators.

```
#!/bin/bash
#
# Update the NIS database maps on a client server. This program
# shouldn’t have to be run, but sometimes things go wrong and we
# have to force a download from the main sever.
#
PATH=/etc/yp:/usr/etc/yp:$PATH

MASTER=myNISserver

for map in auto.direct auto.master ethers.byaddr ethers.byname\
           group.bygid group.byname hosts.byaddr hosts.byname\
           mail.aliases netgroup.byhost netgroup.byuser netgroup\
           netid.byname networks.byaddr networks.byname passwd.byname\
           passwd.byuid priss.byname protocols.byname protocols.bynumber\
           rpc.bynumber services.byname services usenetgroups.byname;
do
  ypxfr $1 -h $MASTER $map
done
```

Procedures and traps

One of the worthy features of the Bourne shell is that it allows you to define *subroutines* or *procedures*. Subroutines work just like subroutines in any other programming language. They are executed in same shell (not as a sub-process).

Here is an interesting program which demonstrates two useful things at the same time. First of all, it shows how to make a hierarchical subroutine structure using the Bourne shell. Secondly, it shows how the `‘trap’` directive can be used to trap signals, so that Bourne shell programs can exit safely when they are killed or when CTRL-C is typed.

```
#!/bin/bash
#
#  How to make a signal handler in Bourne Shell
#  using subroutines
#

#####################################################
# Level 2
#####################################################

ReallyQuit()
{
while true
do
  echo "Do you really want to quit?"
  read answer

  case $answer in

     y* | Y* ) return 0;;
     *)        echo "Resuming..."
               return 1;;

  esac

done
}

#####################################################
# Level 1
```

```
#####################################################

SignalHandler()

{
if ReallyQuit           # Call a function
then
   exit 0
else
   return 0
fi
}

#####################################################
# Level 0 : main program
#####################################################

trap SignalHandler 2 15  # Trap kill signals 2 and 15

echo "Type some lines of text..."

while read text
do

   echo "$text - CTRL-C to exit"

done
```

Note that the logical tree structure of this program is upside down (the highest level comes at the bottom). This is because all subroutines must be defined before they are used.

This example concludes our survey of Bash and the Bourne shell.

setuid and setgid scripts

The superuser `'root'` is the only privileged user in UNIX. All other users have only restricted access to the system. Usually this is desirable, but sometimes it is a nuisance.

A setuid script is a script which has its *setuid-bit* set. When such a script is executed by a user, it is run with all the rights and privileges of the owner of the script. All of the commands in the script are executed as the owner of the file and not with the user-id of the person who ran the script. If the owner of the setuid script is `'root'` then the commands in the script are run with *root privileges*!

Setuid scripts are clearly a touchy security issue. When giving away one's rights to another user (especially those of `'root'`) one is tempting hackers. Setuid scripts should be *avoided*.

A setgid program is almost the same, but only the group id is set to that of the owner of the file. Often the effect is the same.

An example of a setuid program is the `'ps'` program. `'ps'` lists all of the processes running in the kernel. In order to do this it needs permission to access the private data structures in the kernel. By making `'ps'` setgid root, it allows ordinary users to be able to read as much as the writers of `'ps'` thought fit, but no more.

Naturally, only the superuser can make a file setuid or setgid root.

Exercises

1. Write an improved `‘which’` command in Bash.
2. Make a counter program which records in a file how many times you log in to your account. You can call this in your .bashrc file.
3. Make a Bourne shell script to kill all the processes owned by a particular user. (Note, that if you are not the superuser, you cannot kill processes owned by other users.)
4. Write a script to replace the `‘rm’` command with something safer. Think about a way of implementing `‘rm’` so that it is possible to get deleted files back again in case of emergencies. This is not possible using the normal `‘rm’` command. Hint: save files in a hidden directory `‘.deleted’`. Make your script delete files in the `‘.deleted’` directory if they are older than a week, so that you don’t fill up the disk with rubbish.
5. Suppose you have a bunch of files with a particular file-extension: write a script in Bash to change the extension to something else. e.g. to change *.C into *.c. Give the old and new extensions as arguments to the script.
6. Write a program in Bash to search for files in the current directory which contain a certain string. e.g. search for all files which contain the word "if". Hint: use the "find" command.
7. Use the manual pages to find out about the commands `‘at’`, `‘batch’` and `‘atq’`. Test these commands by executing the shell command `‘date’` at some time of your choice. Use the `‘-m’` option so that the result of the job is mailed to you.
8. Write a script in Bash to list all of the files bigger than a certain size starting from the current directory, and including all subdirectories. This kind of program is useful for system administrators when a disk becomes full.

C shell

Programmers who are used to C or C++ often find it easier to program in C-shell because there are strong similarities between the two.

.cshrc and .login files

Most users run the C-shell `‘/bin/csh’` as their login environment, or these days, preferably the `‘tcsh’` which is an improved version of csh. When a user logs in to a UNIX system the C-shell starts by reading some files which configure the environment by defining variables like `path`.

- The file `‘.cshrc’` is searched for in your home directory. i.e. `‘~/.cshrc’.` If it is found, its contents are interpreted by the C-shell as C-shell instructions, before giving you the command prompt(5).
- If and only if this is the *login shell* (not a sub-shell that you have started after login) then the file `‘~/.login’` is searched for and executed.

With the advent of the X11 windowing system, this has changed slightly. Since the window system takes over the entire login procedure, users never get to run ‘login shells’, since the login shell is used up by the X11 system. On an X-terminal or host running X the `'.login'` file normally has no effect.

With some thought, the `'.login'` file can be eliminated entirely, and we can put everything into the `.cshrc` file. Here is a very simple example `'.cshrc'` file.

```
#
# .cshrc - read in by every csh that starts.
#

# Set the default file creation mask
umask 077

# Set the path
set path=( /usr/local/bin /usr/bin/X11 /usr/ucb /bin /usr/bin . )

# Exit here if the shell is not interactive
if ( $?prompt == 0 ) exit

# Set some variables

set noclobber notify filec nobeep
set history=100
set prompt="`hostname`%"
set prompt2 = "%m %h>"    # tcsh, prompt for foreach and while

setenv PRINTER myprinter
setenv LD_LIBRARY_PATH /usr/lib:/usr/local/lib:/usr/openwin/lib

# Aliases are shortcuts to UNIX commands

alias passwd  yppasswd
alias dir     'ls -lg \!* | more'
alias sys     'ps aux | more'
alias h       history
```

It is possible to make a much more complicated .cshrc file than this. The advent of distributed computing and NFS (Network file system) means that you might log into many different machines running different versions of UNIX. The command path would have to be set differently for each type of machine.

Defining variables with set, setenv

We have already seen in the examples above how to define variables in C-shell. Let's formalize this. To define a local variable -- that is, one which will not get passed on to programs and sub-shells running under the current shell, we write

```
set local = "some string"
set myname = "`whoami`"
```

These variables are then referred to by using the dollar `'$'` symbol. i.e. The value of the variable `'local'` is `'$local'`.

```
echo $local $myname
```

Global variables, that is variables which all sub-shells inherit from the current shell are defined using ‘setenv’

```
setenv GLOBAL "Some other string"
setenv MYNAME "‘who am i‘"
```

Their values are also referred to using the ‘$’ symbol. Notice that `set` uses an ‘=’ sign while ‘setenv’ does not.

Variables can be also created without a value. The shell uses this method to switch on and off certain features, using variables like ‘noclobber’ and ‘noglob’. For instance

```
nexus% set flag
nexus% if ($?flag) echo 'Flag is set!'
Flag is set!
nexus% unset flag
nexus% if ( $?flag ) echo 'Flag is set!'
nexus%
```

The operator ‘$?*variable*’ is ‘true’ if *variable* exists and ‘false’ if it does not. It does not matter whether the variable holds any information.

The commands ‘unset’ and ‘unsetenv’ can be used to undefine or delete variables when you don’t want them anymore.

Arrays

A useful facility in the C-shell is the ability to make arrays out of strings and other variables. The round parentheses ‘(..)’ do this. For example, look at the following commands.

```
nexus% set array = ( a b c d )
nexus% echo $array[1]
a
nexus% echo $array[2]
b
nexus% echo $array[$#array]
d

nexus% set noarray = ( "a b c d" )
nexus% echo $noarray[1]
a b c d
nexus% echo $noarray[$#noarray]
a b c d
```

The first command defines an array containing the elements ‘a b c d’. The elements of the array are referred to using square brackets ‘[..]’ and the first element is ‘$array[1]’. The last element is ‘$array[4]’. *NOTE: this is not the same as in C or C++ where the first element of the array is the zeroth element!*

The special operator ‘$#’ returns the number of elements in an array. This gives us a simple way of finding the end of the array. For example

```
nexus% echo $#path
23
```

```
nexus% echo "The last element in path is $path[$#path]"
The last element in path is .
```

To find the next last element we need to be able to do arithmetic. We'll come back to this later.

Pipes and redirection in csh

The symbols

```
              <  >  >>  <<  |  &
```

have a special meaning in the shell. By default, most commands take their input from the file `'stdin'` (the keyboard) and write their output to the file `'stdout'` and their error messages to the file `'stderr'` (normally, both of these output files are defined to be the current terminal device `'/dev/tty'`, or `'/dev/console'`).

`'stdin'`, `'stdout'` and `'stderr'`, known collectively as `'stdio'`, can be redefined or *redirected* so that information is taken from or sent to a different file. The output direction can be changed with the symbol `'>'`. For example,

```
echo testing > myfile
```

produces a file called `'myfile'` which contains the string 'testing'. The single `'>'` (greater than) sign always creates a new file, whereas the double `'>>'` appends to the end of a file, if it already exists. So the first of the commands

```
echo blah blah >> myfile
echo Newfile > myfile
```

adds a second line to `'myfile'` after 'testing', whereas the second command writes over `'myfile'` and ends up with just one line 'Newfile'.

Now suppose we mistype a command

```
ehco test > myfile
```

The command `'ehco'` does not exist and so the error message `'ehco: Command not found'` appears on the terminal. This error message was sent to *stderr* -- so even though we redirected output to a file, the error message appeared on the screen to tell us that an error occurred. Even this can be changed. `'stderr'` can also be redirected by adding an ampersand `'&'` character to the `'>'` symbol. The command

```
ehco test >& myfile
```

results in the file `'myfile'` being created, containing the error message 'ehco: Command not found'.

The input direction can be changed using the `'<'` symbol for example

```
/bin/mail mark < message
```

would send the file `'message'` to the user `'mark'` by electronic mail. The mail program takes its input from the file instead of waiting for keyboard input.

There are some refinements to the redirection symbols. First of all, let us introduce the C-shell variable `‘noclobber’`. If this variable is set with a command like

```
set noclobber
```

then files will not be overwritten by the `‘>’` command. If one tries to redirect output to an existing file, the following happens.

```
UNIX% set noclobber
UNIX% touch blah        # create an empty file blah
UNIX% echo test > blah
blah: File exists.
```

If you are nervous about overwriting files, then you can set `‘noclobber’` in your `‘.cshrc’` file. `‘noclobber’` can be overridden using the pling `‘!’` symbol. So

```
UNIX% set noclobber
UNIX% touch blah        # create an empty file blah
UNIX% echo test >! blah
```

writes over the file `‘blah’` even though `‘noclobber’` is set.

Here are some other combinations of redirection symbols

`‘>>’` Append, including `‘stderr’`
`‘>>!’`
    Append, ignoring `‘noclobber’`
`‘>>&!’`
    Append `‘stdout’`, `‘stderr’`, ignore `‘noclobber’`
`‘<<’` See below.

The last of these commands reads from the standard input until it finds a line which contains a word. It then feeds all of this input into the program concerned. For example,

```
nexus% mail mark <<quit
nexus 1> Hello mark
nexus 2> Nothing much to say...
nexus 2> so bye
nexus 2>
nexus 2> quit
Sending mail...
Mail sent!
```

The mail message contains all the lines up to, but not including ‘marker’. This method can also be used to print text verbatim from a file without using multiple echo commands. Inside a script one may write:

```
cat << "marker";

           MENU

    1) choice 1
    2) choice 2
    ...
marker
```

The `cat` command writes directly to `stdout` and the input is redirected and taken directly from the script file.

A very useful construction is the 'pipe' facility. Using the `'|'` symbol one can feed the `'stdout'` of one program straight into the `'stdin'` of another program. Similarly with `'|&'` both `'stdout'` and `'stderr'` can be piped into the input of another program. This is very convenient. For instance, look up the following commands in the manual and try them.

```
ps aux | more
echo 'Keep on sharpening them there knives!' | mail henry
vmstat 1 | head
ls -l /etc | tail
```

Note that when piping both standard input and standard error to another program, the two files *do not mix synchronously*. Often `'stderr'` appears first.

`'tee'` and `'script'`

Occasionally you might want to have a copy of what you see on your terminal sent to a file. `'tee'` and `'script'` do this. For instance,

```
find / -type l  -print | tee myfile
```

sends a copy of the output of `'find'` to the file 'myfile'. `'tee'` can split the output into as many files as you want:

```
command | tee file1 file2 ....
```

You can also choose to record the output an entire shell session using the `'script'` command.

```
nexus% script mysession
Script started, file is mysession

nexus% echo Big brother is scripting you
Big brother is scripting you

nexus% exit
exit
Script done, file is mysession
```

The file 'mysession' is a text file which contains a transcript of the session.

Scripts with arguments

One of the useful features of the shell is that you can use the normal UNIX commands to make programs called *scripts*. To make a script, you just create a file containing shell commands you want to execute and make sure that the first line of the file looks like the following example.

```
#!/bin/csh -f
#
# A simple script: check for user's mail
#
#

set path = ( /bin /usr/ucb )              # Set the local path

cd /var/spool/mail                        # Change dir

foreach uid ( * )

   echo "$uid has mail in the intray! "    # space prevents an error!

end
```

The sequence `'#!/bin/csh'` means that the following commands are to be fed into `'/bin/csh'`. The two symbols `'#!'` must be the very first two characters in the file. The `'-f'` option means that your `'.cshrc'` file is not read by the shell when it starts up. The file containing this script must be executable (see `'chmod'`) and must be in the current path, like all other programs.

Like C programs, C-shell scripts can accept command line arguments. Suppose you want to make a program to say hello to some other users who are logged onto the system.

```
say-hello mark sarah mel
```

To do this you need to know the names that were typed on the command line. These names are copied into an array in the C-shell called the *argument vector*, or `'argv'`. To read these arguments, you just treat `'argv'` as an array.

```
#!/bin/csh -f
#
# Say hello
#

foreach name ( $argv )

   echo Saying hello to $name
   echo "Hello from $user! " | write $name

end
```

The elements of the array can be referred to as `'argv[1]'`..`'argv[$#argv]'` as usual. They can also be referred to as `'$1'`..`'$3'` up to the last acceptable number. This makes C-shell compatible with the Bourne shell as far as arguments are concerned. One extra flourish in this method is that you can also refer to the name of the program itself as `'$0'`. For example,

```
#!/bin/csh -f

echo This is program $0 running for $user
```

`'$argv'` represents all the arguments. You can also use `'$*'` from the Bourne shell.

Sub-shells ()

The C-shell does not allow you to define subroutines or functions, but you can create a local shell, with its own private variables by enclosing commands in parentheses.

```
#!/bin/csh

cd /etc

( cd /usr/bin; ls * ) > myfile

pwd
```

This program changes the working directory to /etc and then executes a subshell which *inside the brackets* changes directory to /usr/bin and lists the files there. The output of this private shell are sent to a file ‘myfile’. At the end we print out the current working directory just to show that the `‘cd’` command in brackets had no effect on the main program.

Normally both parentheses must be on the same line. If a subshell command line gets too long, so that the brackets are not on the same line, you have to use backslash characters to continue the lines,

```
(  command \
   command \
   command \
)
```

Tests and conditions

No programming language would be complete without tests and loops. C-shell has two kinds of decision structure: the `‘if..then..else’` and the `‘switch’` structure. These are closely related to their C counterparts. The syntax of these is

```
if (condition) command

if (condition) then
   command
   command..
else
   command
   command..
endif


switch (string)

  case one:
            commands
            breaksw

  case two:
            commands
            breaksw
  ...
endsw
```

In the latter case, no commands should appear on the same line as a ‘case’ statement, or they will be ignored. Also, if the `‘breaksw’` commands are omitted, then control flows through all the commands for case 2, case 3 etc, exactly as it does in the C programming language.

We shall consider some examples of these statements in a moment, but first it is worth listing some important tests which can be used in `‘if’` questions to find out information about files.

`‘-r` *`file`*`’`
    True if the file exists and is readable
`‘-w` *`file`*`’`
    True if the file exists and is writable
`‘-x` *`file`*`’`
    True if the file exists and is executable
`‘-e` *`file`*`’`
    True if the file simply exists
`‘-z` *`file`*`’`
    True if the file exists and is empty
`‘-f` *`file`*`’`
    True if the file is a plain file
`‘-d` *`file`*`’`
    True if the file is a directory

We shall also have need of the following comparison operators.

`‘==’`	is equal to (string comparison)
`‘!=’`	is not equal to
`‘>’`	is greater than
`‘<’`	is less than
`‘>=’`	is greater than or equal to
`‘<=’`	is less than or equal to
`‘=~’`	matches a wildcard
`‘!~’`	does not match a wildcard

The simplest way to learn about these statements is to use them, so we shall now look at some examples.

```
#!/bin/csh -f
#
#  Safe copy from <arg[1]> to <arg[2]>
#
#

if ($#argv != 2) then

  echo "Syntax: copy <from-file> <to-file>"
  exit 0
```

```
endif

if ( -f $argv[2] ) then

   echo "File exists. Copy anyway?"

   switch ( $< )                     # Get a line from user

      case y:
               breaksw

      default:
               echo "Doing nothing!"
               exit 0

   endsw

endif

echo -n "Copying $argv[1] to $argv[2]..."
cp $argv[1] $argv[2]
echo done

endif
```

This script tries to copy a file from one location to another. If the user does not type exactly two arguments, the script quits with a message about the correct syntax. Otherwise it tests to see whether a plain file has the same name as the file the user wanted to copy to. If such a file exists, it asks the user if he/she wants to continue before proceeding to copy.

Switch example: configure script

Here is another example which compiles a software package. This is a problem we shall return to later See section Make. The problem this script tries to address is the following. There are many different versions of UNIX and they are not exactly compatible with one another. The program this file compiles has to work on any kind of UNIX, so it tries first to determine what kind of UNIX system the script is being run on by calling `‘uname’`. Then it defines a variable `‘MAKE’` which contains the path to the ‘make’ program which will build *software*. The make program reads a file called ‘Makefile’ which contains instructions for compiling the program, but this file needs to know the type of UNIX, so the script first copies a file ‘Makefile.src’ using `‘sed’` replace a dummy string with the real name of the UNIX. Then it calls make and sets the correct permission on the file using `‘chmod’`.

```
#!/bin/csh -f
#################################################
#
#
# CONFIGURE Makefile AND BUILD software
#
#
#################################################

set NAME = ( ‘uname -r -s‘ )

switch ($NAME[1])

   case SunOS*:
```

```
                  switch ($NAME[2])

                     case 4*:
                              setenv TYPE SUN4
                              setenv MAKE /bin/make
                              breaksw
                     case 5*:
                              setenv TYPE SOLARIS
                              setenv MAKE /usr/ccs/bin/make
                              breaksw

                  endsw
                  breaksw

   case ULTRIX*:
                  setenv TYPE ULTRIX
                  setenv MAKE /bin/make
                  breaksw
   case HP-UX*:
                  setenv TYPE HPUX
                  setenv MAKE /bin/make
                  breaksw
   case AIX*:
                  setenv TYPE AIX
                  setenv MAKE /bin/make
                  breaksw

   case OSF*:
                  setenv TYPE OSF
                  setenv MAKE /bin/make
                  breaksw
   case IRIX*:
                  setenv TYPE IRIX
                  setenv MAKE /bin/make
                  breaksw

   default:
                  echo Unknown architecture $NAME[1]

endsw

 # Generate Makefile from source file

sed s/HOSTTYPE/$TYPE/ Makefile.src > Makefile

echo "Making software. Type CTRL-C to abort and edit Makefile"

$MAKE software         # call make to build program
chmod 755 software     # set correct protection
```

Loops in csh

The C-shell has three loop structures: `‘repeat’`, `‘while’` and `‘foreach’`. We have already seen some examples of the `‘foreach’` loop.

The structure of these loops is as follows

`repeat` *`number-of-times command`*

```
while ( test expression )

   commands

end

foreach  control-variable  ( list-or-array )

   commands

end
```

The commands `‘break’` and `‘continue’` can be used to break out of the loops at any time. Here are some examples.

```
repeat 2 echo "Yo!" | write mark
```

This sends the message "Yo!" to mark’s terminal twice.

```
repeat 5 echo ‘echo "Shutdown time! Log out now" | wall ; sleep 30‘ ; halt
```

This example repeats the command ‘echo Shutdown time...’ five times at 30 second intervals, before shutting down the system. Only the superuser can run this command! Note the strange construction with ‘echo echo’. This is to force the repeat command to take two shell commands as an argument. (Try to explain why this works for yourself.)

Input from the user

```
# Test a user response

echo "Answer y/n (yes or no)"

set valid = false

while ( $valid == false )

   switch ( $< )

      case y:
              echo "You answered yes"
              set valid = true
              breaksw

      case n:
              echo "You answered no"
              set valid = true
              breaksw

      default:
              echo "Invalid response, try again"
              breaksw

   endsw
```

```
end
```

Notice that it would have been simpler to replace the two lines

```
  set valid = true
  breaksw
```

by a single line `‘break’`. `‘breaksw’` jumps out of the switch construction, after which the `‘while’` test fails. `‘break’` jumps out of the entire while loop.

Extracting parts of a pathname

A path name consists of a number of different parts:

- The path to the directory where a file is held.
- The name of the file itself.
- The file extension (after a dot).

By using one of the following modifiers, we can extract these different elements.

`‘:h’`	The path to the file
`‘:t’`	The filename itself
`‘:e’`	The file extension
`‘:r’`	The complete file-path minus the file extension

Here are some examples and the results:

```
set f = ~/progs/c++/test.C

echo $f:h

 /home/mark/progs/c++

echo $f:t

  test.C

echo $f:e

   C
echo $f:r

   /home/mark/progs/c++/test
```

Arithmetic

Before using these features in a real script, we need one more possibility: numerical addition, subtraction and multiplication etc.

To tell the C-shell that you want to perform an operation on numbers rather than strings, you use the `‘@’` symbol followed by a space. Then the following operations are possible.

```
@ var = 45                        # Assign a numerical value to var
echo $var                         # Print the value

@ var = $var + 34                 # Add 34 to var
@ var += 34                       # Add 34 to var

@ var -= 1                        # subtract 1 from var
@ var *= 5                        # Multiply var by 5

@ var /= 3                        # Divide var by 3 (integer division)
@ var %= 3                        # Remainder after dividing var by 3

@ var++                           # Increment var by 1
@ var--                           # Decrement var by 1

@ array[1] = 5                    # Numerical array

@ logic = ( $x > 6 && $x < 10)    # AND
@ logic = ( $x > 6 || $x < 10)    # OR
@ false = ! $var                  # Logical NOT

@ bits = ( $x | $y )              # Bitwise OR
@ bits = ( $x ^ $y )              # Bitwise XOR
@ bits = ( $x & $y )              # Bitwise AND

@ shifted = ( $var >> 2 )         # Bitwise shift right
@ back    = ( $var << 2 )         # Bitwise shift left
```

These operators are precisely those found in the C programming language.

Examples

The following script uses the operators in the last two sections to take a list of files with a given file extension (say `'.doc'`) and change it for another (say `'.tex'`). This is a partial solution to the limitation of not being able to do multiple renames in shell.

```
#!/bin/csh -f
#############################################################
#
# Change file extension for multiple files
#
#############################################################

if ($#argv < 2) then
  echo Syntax: chext oldpattern newextension
  echo "e.g: chext *.doc tex "
  exit 0
endif

mkdir /tmp/chext.$user                 # Make a scratch area

set newext="$argv[$#argv]"             # Last arg is new ext
set oldext="$argv[1]:e"

echo "Old extension was ($oldext)""
echo "New extension ($newext) -- okay? (y/n)"

switch( $< )
```

```
   case y:
           breaksw
   default:
           echo "Nothing done."
           exit 0
endsw

##############################################################
# Remove the last file extension from files
##############################################################

 i = 0

foreach file ($argv)

   i++
  if ( $i == $#argv ) break
  cp $file /tmp/chext.$user/$file:r        # temporary store

end

###############################################################
# Add .newext file extension to files
###############################################################

set array = (‘ls /tmp/chext.$user‘)

foreach file ($array)

  if ( -f $file.$newext ) then
    echo  destination file $file.$newext exists. No action taken.
    continue
  endif

  cp /tmp/chext.$user/$file $file.$newext
  rm $file.$oldext

end

rm -r /tmp/chext.$user
```

Here is another example to try to decipher. Use the manual pages to find out about `‘awk’`. This script can be written much more easily in Perl or C, as we shall see in the next chapters. It is also trivially implemented as a script in the system administration language cfengine.

```
#!/bin/csh -f
###########################################################
#
# KILL all processes owned by $argv[1] with PID > $argv[2]
#
###########################################################


if ("‘whoami‘" != "root") then
  echo Permission denied
  exit 0
endif

if ( $#argv < 1 || $#argv > 2 ) then
```

```
  echo Usage: KILL username lowest-pid
  exit 0
endif

if ( $argv[1] == "root") then
  echo No! Too dangerous -- system will crash
  exit 0
endif

############################################################
# Kill everything
############################################################

if ( $#argv == 1 ) then

  set killarray = ( ‘ps aux |  awk ’{ if ($1 == user) \
{printf "%s ",$2}}’ user=$argv[1]‘ )

  foreach process ($killarray)

     kill -1 $process
     kill -15 $process > /dev/null
     kill -9 $process > /dev/null

     if ("‘kill -9 $process | egrep -e ’No such process’‘" == "") then
        echo "Warning - $process would not die - try again"
     endif
  end

#############################################################
# Start from a certain PID
#############################################################

else if ( $#argv == 2 ) then

  set killarray = ( ‘ps aux |  awk ’{ if ($1 == user && $2 > uid) \
{printf "%s ",$2}}’ user=$argv[1] uid=$argv[2]‘ )

  foreach process ($killarray)

     kill -1 $process > /dev/null
     kill -15 $process
     sleep 2
     kill -9 $process > /dev/null

     if ("‘kill -9 $process | egrep -e ’No such process’‘" == "") then
        echo "Warning - $process would not die - try again"
     endif
  end

endif
```

This program would be better written in C or Perl.

Summary: Limitations of shell programming

To summarize the last two long and oppressive chapters we shall take a step back from the details and look at what we have achieved.

The idea behind the shell is to provide a user interface, with access to the system's facilities at a simple level. In the 70's user interfaces were not designed to be user-friendly. The UNIX shell is not particularly use friendly, but it is very powerful. Perhaps it would have been enough to provide only commands to allow users to write C programs. Since all of the system functions are available from C, that would certainly allow everyone to do what anything that UNIX can do. But shell programming is much more *immediate* than C. It is an environment of *frequently used tools*. Also for quick programming solutions: C is a compiled language, whereas the shell is an interpreter. A quick shell program can solve many problems in no time at all, without having to compile anything.

Shell programming is only useful for 'quick and easy' programs. To use it for anything serious is an abuse. Programming difficult things in shell is clumsy, and it is difficult to get returned-information (like error messages) back in a useful form. Besides, shell scripts are slow compared to real programs since they involve starting a new program for each new command.

These difficulties are solved partly by Perl, which we shall consider next -- but in the final analysis, real programs of substance need to be written in C. Contrary to popular belief, this is not more difficult than programming in the shell -- in fact, many things are much simpler, because all of the shell commands originated as C functions. The shell is an extra layer of the UNIX onion which we have to battle our way through to get where we're going.

Sometimes it is helpful to be shielded from *low level* details -- sometimes it is a *hindrance*. In the remaining chapters we shall consider more involved programming needs.

Perl

So far, we have been looking at shell programming for performing fairly simple tasks. Now let's extend the idea of shell programming to cover more complex tasks like systems programming and network communications. Perl is a language which was designed to retain the immediateness of shell languages, but at the same time capture some of the flexibility of C. Perl is an acronym for *Practical extraction and report language*. In this chapter, we shall not aim to teach Perl from scratch -- the best way to learn it is to use it! Rather we shall concentrate on demonstrating some principles.

Sed and awk, cut and paste

One of the reasons for using Perl is that it is extremely good at textfile handling--one of the most important things for UNIX users, and particularly useful in connection with CGI script processing on the World Wide Web. It has simple built-in constructs for searching and replacing text, storing information in arrays and retrieving them in sorted form. All of the these things have previously been possible using the UNIX shell commands

```
sed
awk
cut
paste
```

but these commands were designed to work primarily in the Bourne shell and are a bit `'awk'`ward to use for all but the simplest applications.

`‘sed’`
: is a stream editor. It takes command line instructions, reads input from the stream `stdin` and produces output on *stdout* according to those instructions. `‘sed’` works line by line from the start of a textfile.

`‘awk’`
: is a pattern matching and processing language. It takes a textfile and reads it line by line, matching *regular expressions* and acting on them. `‘awk’` is powerful enough to have conditional instructions like `‘if..then..else’` and uses C’s `‘printf’` construction for output.

`‘cut’`
: Takes a line of input and cuts it into *fields*, separated by some character. For instance, a normal line of text is a string of words separated by spaces. Each word is a different field. `‘cut’` can be used, for instance, to pick out the third column in a table. Any character can be specified as the separator.

`‘paste’`
: is the logical opposite of cut. It concatenates @math{n} files, and makes each line in the file into a column of a table. For instance, `‘paste one two three’` would make a table in which the first column consisted of all lines in `‘one’`, the second of all lines in `‘two’` and the third of all lines in `‘three’`. If one file is longer than the others, then some columns have blank spaces.

Perl unifies all of these operations and more. It also makes them much simpler.

Program structure

To summarize Perl, we need to know about the structure of a Perl program, the conditional constructs it has, its loops and its variables. In the latest versions of Perl (Perl 5), you can write object oriented programs of great complexity. We shall not go into this depth, for the simple reason that Perl’s strength is not as a general programming language but as a specialized language for textfile handling. The syntax of Perl is in many ways like the C programming language, but there are important differences.

- Variables do not have *types*. They are interpreted in a context sensitive way. The operators which acts upon variables determine whether a variable is to be considered a string or as an integer etc.
- Although there are no types, Perl defines *arrays* of different kinds. There are three different kinds of array, labelled by the symbols `‘$’`, `‘@’` and `‘%’`.
- Perl keeps a number of standard variables with special names e.g. `‘$_ @ARGV’` and `‘%ENV’`. Special attention should be paid to these. They are very important!
- The shell reverse apostrophe notation `‘command‘` can be used to execute UNIX programs and get the result into a Perl variable.

Here is a simple ‘structured hello world’ program in Perl. Notice that subroutines are called using the `‘&’` symbol. There is no special way of marking the main program -- it is simply that part of the program which starts at line 1.

```
#!/local/bin/perl
#
# Comments
#

&Hello();
&World;
```

```
# end of main

sub Hello
   {
   print "Hello";
   }

sub World
   {
   print "World\n";
   }
```

The parentheses on subroutines are optional, if there are no parameters passed. Notice that each line must end in a semi-colon.

Perl variables

Scalar variables

In Perl, variables do not have to be declared before they are used. Whenever you use a new symbol, Perl automatically adds the symbol to its symbol table and initializes the variable to the empty string.

It is important to understand that there is no practical difference between zero and the empty string in perl -- except in the way that you, the user, choose to use it. Perl makes no distinction between strings and integers or any other types of data -- except when it wants to interpret them. For instance, to compare two variables as strings is not the same as comparing them as integers, even if the string contains a textual representation of an integer. Take a look at the following program.

```
#!/local/bin/perl
#
# Nothing!
#

print "Nothing == $nothing\n";

print "Nothing is zero!\n" if ($nothing == 0);

if ($nothing eq "")
   {
   print STDERR "Nothing is really nothing!\n";
   }

$nothing = 0;

print "Nothing is now $nothing\n";
```

The output from this program is

```
Nothing ==
Nothing is zero!
Nothing is really nothing!
Nothing is now 0
```

There are several important things to note here. First of all, we never declare the variable ‘nothing’. When we try to write its value, perl creates the name and associates a NULL value to it i.e. the empty string. There is no error. Perl knows it is a variable because of the `‘$’` symbol in front of it. All *scalar* variables are identified by using the dollar symbol.

Next, we compare the value of `‘$nothing’` to the integer ‘0’ using the integer comparison symbol `‘==’`, and then we compare it to the empty string using the string comparison symbol `‘eq’`. Both tests are true! That means that the empty string is interpreted as having a numerical value of zero. In fact *any string* which does not form a valid integer number has a numerical value of zero.

Finally we can set `‘$nothing’` explicitly to a valid integer string zero, which would now pass the first test, but fail the second.

As extra spice, this program also demonstrates two different ways of writing the `‘if’` command in perl.

The default scalar variable.

The special variable `‘$_’` is used for many purposes in Perl. It is used as a buffer to contain the result of the last operation, the last line read in from a file etc. It is so general that many functions which act on scalar variables work by default on `‘$_’` if no other argument is specified. For example,

```
print;
```

is the same as

```
print $_;
```

Array (vector) variables

The complement of scalar variables is arrays. An array, in Perl is identified by the `‘@’` symbol and, like scalar variables, is allocated and initialized dynamically.

```
@array[0] = "This little piggy went to market";
@array[2] = "This little piggy stayed at home";

print "@array[0] @array[1] @array[2]";
```

The index of an array is always understood to be a number, not a string, so if you use a non-numerical string to refer to an array element, you will always get the zeroth element, since a non-numerical string has an integer value of zero.

An important array which every program defines is

```
@ARGV
```

This is the argument vector array, and contains the commands line arguments by analogy with the C-shell variable `‘$argv[]’`.

Given an array, we can find the last element by using the `'$#'` operator. For example,

```
$last_element = $ARGV[$#ARGV];
```

Notice that each element in an array is a scalar variable. The `'$#'` cannot be interpreted directly as the number of elements in the array, as it can in the C-shell. You should experiment with the value of this quantity -- it often necessary to add 1 or 2 to its value in order to get the behaviour one is used to in the C-shell.

Perl does not support multiple-dimension arrays directly, but it is possible to simulate them yourself. (See the Perl book.)

Special array commands

The `'shift'` command acts on arrays and returns and removes the first element of the array. Afterwards, all of the elements are shifted down one place. So one way to read the elements of an array in order is to repeatedly call `'shift'`.

```
$next_element=shift(@myarray);
```

Note that, if the array argument is omitted, then `'shift'` works on `'@ARGV'` by default.

Another useful function is `'split'`, which takes a string and turns it into an array of strings. `'split'` works by choosing a character (usually a space) to delimit the array elements, so a string containing a sentence separated by spaces would be turned into an array of words. The syntax is

```
@array = split;                       # works with spaces on $_
@array = split(pattern,string);       # Breaks on pattern
($v1,$v2...) = split(pattern,string); # Name array elements with scalars
```

In the first of these cases, it is assumed that the variable `'$_'` is to be split on whitespace characters. In the second case, we decide on what character the split is to take place and on what string the function is to act. For instance

```
@new_array = split(":","name:passwd:uid:gid:gcos:home:shell");
```

The result is a seven element array called `'@new_array'`, where `'$new_array[0]'` is `'name'` etc.

In the final example, the left hand side shows that we wish to capture elements of the array in a named set of scalar variables. If the number of variables on the lefthand side is fewer than the number of strings which are generated on the right hand side, they are discarded. If the number on the left hand side is greater, then the remainder variables are empty.

Associated arrays

One of the very nice features of Perl is the ability to use one string as an index to another string in an array. For example, we can make a short encyclopedia of zoo animals by constructing an associative array in which the keys (or indices) of the array are the names of animals, and the contents of the array are the information about them.

```
$animals{"Penguin"} = "A suspicious animal, good with cheese crackers...";
$animals{"dog"} = "Plays stupid, but could be a cover...";

if ($index eq "fish")
   {
   $animals{$index} = "Often comes in square boxes. Very cold.";
   }
```

An entire associated array is written `‘%array’`, while the elements are `‘$array{$key}’`.

Perl provides a special associative array for every program called `‘%ENV’`. This contains the *environment variables* defined in the parent shell which is running the Perl program. For example

```
print "Username = $ENV{"USER"}\n";

$ld = "LD_LIBRARY_PATH";
print "The link editor path is $ENV{$ld}\n";
```

To get the current path into an ordinary array, one could write,

```
@path_array= split(":",$ENV{"PATH"});
```

Array example program

Here is an example which prints out a list of files in a specified directory, in order of their UNIX protection bits. The *least* protected file files come first.

```
#!/local/bin/perl
#
# Demonstration of arrays and associated arrays.
# Print out a list of files, sorted by protection,
# so that the least secure files come first.
#
# e.g.     arrays <list of words>
#          arrays *.C
#
############################################################

print "You typed in ",$#ARGV+1," arguments to command\n";

if ($#ARGV < 1)
   {
   print "That’s not enough to do anything with!\n";
   }

while ($next_arg = shift(@ARGV))
   {
   if ( ! ( -f $next_arg || -d $next_arg))
      {
      print "No such file: $next_arg\n";
      next;
      }

   ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size) = stat($next_arg);
   $octalmode = sprintf("%o",$mode & 0777);
   $assoc_array{$octalmode} .= $next_arg.
            " : size (".$size."), mode (".$octalmode.")\n";
   }

print "In order: LEAST secure first!\n\n";

foreach $i (reverse sort keys(%assoc_array))
   {
   print $assoc_array{$i};
   }
```

Loops and conditionals

Here are some of the most commonly used decision-making constructions and loops in Perl. The following is not a comprehensive list -- for that, you will have to look in the Perl bible: *Programming Perl*, by Larry Wall and Randal Schwartz. The basic pattern follows the C programming language quite closely. In the case of the `‘for’` loop, Perl has both the C-like version, called `‘for’` and a `‘foreach’` command which is like the C-shell implementation.

```
if (expression)
   {
   block;
   }
else
   {
   block;
   }

command if (expression);

unless (expression)
   {
   block;
   }
else
   {
   block;
   }

while (expression)
   {
   block;
   }

do
   {
   block;
   }
while (expression);

for (initializer; expression; statement)
   {
   block;
   }

foreach variable(array)
   {
```

```
   block;
   }
```

In all cases, the `‘else’` clauses may be omitted.

Strangely, perl does not have a `‘switch’` statement, but the Perl book describes how to make one using the features provided.

The for loop

The for loop is exactly like that in C or C++ and is used to iterate over a numerical index, like this:

```
for ($i = 0; $i < 10; $i++)
   {
   print $i, "\n";
   }
```

The foreach loop

The foreach loop is like its counterpart in the C shell. It is used for reading elements one by one from a regular array. For example,

```
foreach $i ( @array )
   {
   print $i, "\n";
   }
```

Iterating over elements in arrays

One of the main uses for `‘for’` type loops is to iterate over successive values in an array. This can be done in two ways which show the essential difference between `for` and `foreach`.

If we want to fetch each value in an array in turn, without caring about numerical indices, the it is simplest to use the `foreach` loop.

```
 @array = split(" ","a b c d e f g");

 foreach $var ( @array )
    {
    print $var, "\n";
    }
```

This example prints each letter on a separate line. If, on the other hand, we are interested in the index, for the purposes of some calculation, then the `for` loop is preferable.

```
 @array = split(" ","a b c d e f g");
```

```
for ($i = 0; $i <= $#array; $i++)
   {
   print $array[$i], "\n";
   }
```

Notice that, unlike the for-loop idiom in C/C++, the limit is `'$i <= $#array'`, i.e. 'less than or equal to' rather than 'less than'. This is because the `'$#'` operator does not return the number of elements in the array but rather the last element.

Associated arrays are slightly different, since they do not use numerical keys. Instead they use a set of strings, like in a database, so that you can use one string to look up another. In order to iterate over the values in the array we need to get a list of these strings. The `keys` command is used for this.

```
$assoc{"mark"} = "cool";
$assoc{"GNU"} = "brave";
$assoc{"zebra"} = "stripy";

foreach $var ( keys %assoc )
   {
   print "$var , $assoc{$var} \n";
   }
```

The order of the keys is not defined in the above example, but you can choose to sort them alphabetically by writing

```
foreach $var ( sort keys %assoc )
```

instead.

Iterating over lines in a file

Since Perl is about file handling we are very interested in reading files. Unlike C and C++, perl likes to read files line by line. The angle brackets are used for this, See section Files in perl. Assuming that we have some file handle `'<file>'`, for instance `'<STDIN>'`, we can always read the file line by line with a while-loop like this.

```
while ($line = <file>)
   {
   print $line;
   }
```

Note that `$line` includes the end of line character on the end of each line. If you want to remove it, you should add a 'chop' command:

```
while ($line = <file>)
   {
   chop $line;
   print "line = ($line)\n";
   }
```

Files in perl

Opening files is straightforward in Perl. Files must be opened and closed using -- wait for it -- the commands `‘open’` and `‘close’`. You should be careful to close files after you have finished with them -- especially if you are writing *to* a file. Files are buffered and often large parts of a file are not actually written until the `‘close’` command is received.

Three files are, of course, always open for every program, namely `‘STDIN’`, `‘STDOUT’`and `‘STDERR’`.

Formally, to open a file, we must obtain a file descriptor or file handle. This is done using `‘open’`;

```
open (file_descrip,"Filename");
```

The angular brackets `‘<..>’` are used to read from the file. For example,

```
$line = <file_descrip>;
```

reads one line from the file associated with `‘file_descrip’`.

Let’s look at some examples of filing opening. Here is how we can implement UNIX’s `‘cut’` and `‘paste’` commands in perl:

```
#!/local/bin/perl
#
# Cut in perl
#

 # Cut second column

while (<>)
   {
   @cut_array = split;

   print "@cut_array[1]\n";
   }
```

This is the simplest way to open a file. The empty file descriptor `‘<>’` tells perl to take the argument of the command as a filename and open that file for reading. This is really short for `‘while($_=<STDIN>)’` with the standard input redirected to the named file.

The `‘paste’`program can be written as follows:

```
#!/local/bin/perl
#
# Paste in perl
#
# Two files only, syntax : paste file 1file2
#

open (file1,"@ARGV[0]") || die "Can’t open @ARGV[0]\n";
open (file2,"@ARGV[1]") || die "Can’t open @ARGV[1]\n";

while (($line1 = <file1>) || ($line2 = <file2>))
```

```
   {
   chop $line1;
   chop $line2;

   print "$line1        $line2\n";    # tab character between
   }
```

Here we see more formally how to read from two separate files at the same time. Notice that, by putting the read commands into the test-expression for the `‘while’` loop, we are using the fact that `‘<..>’` returns a non-zero (true) value unless we have reached the end of the file.

To write and append to files, we use the shell redirection symbols inside the `‘open’` command.

```
open(fd,"> filename");    # open file for writing
open(fd,">> filename");   # open file for appending
```

We can also open a pipe from an arbitrary UNIX command and receive the output of that command as our input:

```
open (fd,"/bin/ps aux | ");
```

A simple perl program

Let us now write the simplest perl program which illustrates the way in which perl can save time. We shall write it in three different ways to show what the short cuts mean. Let us implement the `‘cat’` command, which copies files to the standard output. The simplest way to write this is perl is the following:

```
#!/local/bin/perl

while (<>)
   {
    print;
   }
```

Here we have made heavy use of the many default assumptions which perl makes. The program is simple, but difficult to understand for novices. First of all we use the default file handle <> which means, take one line of input from a default file. This object returns true as long as it has not reached the end of the file, so this loop continues to read lines until it reaches the end of file. The default file is standard input, unless this script is invoked with a command line argument, in which case the argument is treated as a filename and perl attempts to open the argument-filename for reading. The `print` statement has no argument telling it what to print, but perl takes this to mean: print the default variable `‘$_’`.

We can therefore write this more explicitly as follows:

```
#!/local/bin/perl

open (HANDLE,"$ARGV[1]");

while (<HANDLE>)
```

```
   {
    print $_;
   }
```

Here we have simply filled in the assumptions explicitly. The command `‘<HANDLE>’` now reads a single line from the named file-handle into the default variable `‘$_’`. To make this program more general, we can eliminate the defaults entirely.

```
#!/local/bin/perl

open (HANDLE,"$ARGV[1]");

while ($line=<HANDLE>)
   {
    print $line;
   }
```

== and ‘eq’

Be careful to distinguish between the comparison operator for integers `‘==’` and the corresponding operator for strings `‘eq’`. These do not work in each other’s places so if you get the wrong comparison operator your program might not work and it is quite difficult to find the error.

chop

The command `‘chop’` cuts off the last character of a string. This is useful for removing newline characters when reading files etc. The syntax is

```
chop;         # chop $_;

chop $scalar; # remove last character in $scalar
```

Perl subroutines

Subroutines are indicated, as in the example above, by the ampersand `‘&’` symbol. When parameters are passed to a Perl subroutine, they are handed over as an array called `‘@_’`. Which is analogous to the `‘$_’` variable. Here is a simple example:

```
#!/local/bin/perl

$a="silver";
$b="gold";

&PrintArgs($a,$b);

# end of main

sub PrintArgs

   {
   ($local_a,$local_b) = @_;

   print "$local_a, $local_b\n";
```

```
    }
```

die - exit on error

When a program has to quit and give a message, the `‘die’` command is normally used. If called without an argument, Perl generates its own message including a line number at which the error occurred. To include your own message, you write

```
die "My message....";
```

If the string is terminated with a `‘\n’` newline character, the line number of the error is not printed, otherwise Perl appends the line number to your string.

When opening files, it is common to see the syntax:

```
open (filehandle,"Filename") || die "Can’t open...";
```

The logical `‘OR’` symbol is used, because `‘open’` returns true if all goes well, in which case the right hand side is never evaluated. If `‘open’` is false, then die is executed. You can decide for yourself whether or not you think this is good programming style -- we mention it here because it is common practice.

The `stat()` idiom

The UNIX library function `stat()` is used to find out information about a given file. This function is available both in C and in Perl. In perl, it returns an array of values. Usually we are interested in knowing the access permissions of a file. `stat()` is called using the syntax

```
@array = stat ("filename");
```

or alternatively, using a named array

```
($device,$inode,$mode) = stat("filename");
```

The value returned in the *mode* variable is a bit-pattern, See section Protection bits. The most useful way of treating these bit patterns is to use octal numbers to interpret their meaning.

To find out whether a file is readable or writable to a group of users, we use a programming idiom which is very common for dealing with bit patterns: first we define a mask which zeroes out all of the bits in the mode string except those which we are specifically interested in. This is done by defining a mask value in which the bits we want are set to 1 and all others are set to zero. Then we AND the mask with the mode string. If the result is different from zero then we know that all of the bits were also set in the mode string. As in C, the bitwise AND operator in perl is called `‘&’`.

For example, to test whether a file is writable to other users in the same group as the file, we would write the following.

```
$mask = 020;   # Leading 0 means octal number

($device,$inode,$mode) = stat("file");

if ($mode & $mask)
   {
   print "File is writable by the group\n";
   }
```

Here the 2 in the second octal number means "write", the fact that it is the second octal number from the right means that it refers to "group". Thus the result of the if-test is only true if that particular bit is true. We shall see this idiom in action below.

Perl example programs

The passwd program and `'crypt()'` function

Here is a simple implementation of the UNIX `'passwd'` program in Perl.

```
#!/local/bin/perl
#
# A perl version of the passwd program.
#
# Note - the real passwd program needs to be much more
# secure than this one. This is just to demonstrate the
# use of the crypt() function.
#
##############################################################

print "Changing passwd for $ENV{'USER'} on $ENV{'HOST'}\n";

system 'stty','-echo';
print "Old passwd: ";

$oldpwd = <STDIN>;
chop $oldpwd;

($name,$coded_pwd,$uid,$gid,$x,$y,$z,$gcos,$home,$shell)
                                 = getpwnam($ENV{"USER"});

if (crypt($oldpwd,$coded_pwd) ne $coded_pwd)
   {
   print "\nPasswd incorrect\n";
   exit (1);
   }

$oldpwd = "";                         # Destroy the evidence!

print "\nNew passwd: ";

$newpwd = <STDIN>;

print "\nRepeat new passwd: ";
```

```
$rnewpwd = <STDIN>;

chop $newpwd;
chop $rnewpwd;

if ($newpwd ne $rnewpwd)
   {
   print "\n Incorrectly typed. Password unchanged.\n";
   exit (1);
   }

$salt = rand();
$new_coded_pwd = crypt($newpwd,$salt);

print "\n\n$name:$new_coded_pwd:$uid:$gid:$gcos:$home:$shell\n";
```

Example with `'fork()'`

The following example uses the `'fork'` function to start a daemon which goes into the background and watches the system to which process is using the greatest amount of CPU time each minute. A pipe is opened from the BSD `'ps'` command.

```
#!/local/bin/perl
#
# A fork() demo. This program will sit in the background and
# make a list of the process which uses the maximum CPU average
# at 1 minute intervals. On a quiet BSD like system this will
# normally be the swapper (long term scheduler).
#

$true = 1;
$logfile="perl.cpu.logfile";

print "Max CPU logfile, forking daemon...\n";

if (fork())
   {
   exit(0);
   }

while ($true)
   {
   open (logfile,">> $logfile") || die "Can't open $logfile\n";
   open (ps,"/bin/ps aux |") || die "Couldn't open a pipe from ps !!\n";

   $skip_first_line = <ps>;
   $max_process = <ps>;
   close(ps);

   print logfile $max_process;
   close(logfile);
   sleep 60;

   ($a,$b,$c,$d,$e,$f,$g,$size) = stat($logfile);

   if ($size > 500)
      {
      print STDERR "Log file getting big, better quit!\n";
      exit(0);
      }
   }
```

Example reading databases

Here is an example program with several of the above features demonstrated simultaneously. This following program lists all users who have home directories on the current host. If the home area has sub-directories, corresponding to groups, then this is specified on the command line. The word ‘home’ causes the program to print out the home directories of the users.

```
#!/local/bin/perl
##################################################################
#
# allusers - list all users on named host, i.e. all
#            users who can log into this machine.
#
# Syntax: allusers group
#         allusers mygroup home
#         allusers myhost group home
#
# NOTE : This command returns only users who are registered on
#        the current host. It will not find users which cannot
#        be validated in the passwd file, or in the named groups
#        in NIS. It assumes that the users belonging to
#        different groups are saved in subdirectories of
#        /home/hostname.
#
##################################################################

&arguments();

die "\n" if ( ! -d "/home/$server" );

$disks = ‘/bin/ls -d /home/$server/$group‘;

foreach $home (split(/\s/,$disks))
   {
   open (LS,"cd $home; /bin/ls $home |") || die "allusers: Pipe didn’t open";

   while (<LS>)
      {
      $exists = "";
      ($user) = split;
      ($exists,$pw,$uid,$gid,$qu,$cm,$gcos,$dir)=getpwnam($user);

      if ($exists)
         {
         if ($printhomes)
            {
            print "$dir\n";
            }
         else
            {
            print "$user\n";
            }
         }
      }
   close(LS);
   }

########################################################

sub arguments
   {
   $printhomes = 0;
   $group = "*";
   $server = ‘/bin/hostname‘;
   chop $server;

   foreach $arg (@ARGV)
      {
      if (substr($arg,0,1) eq "u")
         {
         $group = $arg;
         next;
         }

      if ($arg eq "home")
         {
         $printhomes = 1;
         next;
         }

      $server= $arg;     #default is to interpret as a server.
      }
   }
```

Pattern matching and extraction

Perl has regular expression operators for identifying patterns. The operator

```
     /regular expression/
```

returns true of false depending on whether the regular expression matches the contents of `$_`. For example

```
  if (/perl/)
     {
     print "String contains perl as a substring";
     }

  if (/(Sat|Sun)day/)
     {
     print "Weekend day....";
     }
```

The effect is rather like the `grep` command. To use this operator on other variables you would write:

```
  $variable =~ /regexp/
```

Regular expression can contain parenthetic sub-expressions, e.g.

```
  if (/(Sat|Sun)day (..)th (.*)/)
     {
     $first = $1;
     $second = $2;
     $third = $3;
     }
```

in which case perl places the objects matched by such sub-expressions in the variables `$1`, `$2` etc.

Searching and replacing text

The ‘`sed`’-like function for replacing all occurances of a string is easily implemented in Perl using

```
while (<input>)
   {
   s/$search/$replace/g;
   print output;
   }
```

This example replaces the string inside the default variable. To replace in a general variable we use the operator ‘`=~`’ with syntax:

```
$variable =~ s/search/replace/
```

Here is an example of some of this operator in use. The following is a program which searches and replaces a string in several files. This is useful program indeed for making a change globally in a group of files! The program is called ‘file-replace’.

```
#!/local/bin/perl
##############################################################
#
# Look through files for findstring and change to newstring
# in all files.
#
##############################################################

#
# Define a temporary file and check it doesn’t exist
#

$outputfile = "tmpmarkfind";
unlink $outputfile;

#
# Check command line for list of files
#

if ($#ARGV < 0)
   {
   die "Syntax: file-replace [file list]\n";
   }

print "Enter the string you want to find (Don’t use quotes):\n\n:";
```

```
$findstring=<STDIN>;
chop $findstring;

print "Enter the string you want to replace with (Don’t use quotes):\n\n:";
$replacestring=<STDIN>;
chop $replacestring;

#

print "\nFind: $findstring\n";
print "Replace: $replacestring\n";
print "\nConfirm (y/n)  ";
$y = <STDIN>;
chop $y;

if ( $y ne "y")
   {
   die "Aborted -- nothing done.\n";
   }
else
   {
   print "Use CTRL-C to interrupt...\n";
   }

#
# Now shift default array @ARGV to get arguments 1 by 1
#

while ($file = shift)
   {
   if ($file eq "file-replace")
      {
      print "Findmark will not operate on itself!";
      next;
      }

   #
   # Save existing mode of file for later
   #

   ($dev,$ino,$mode)=stat($file);

   open (INPUT,$file) || warn "Couldn’t open $file\n";
   open (OUTPUT,"> $outputfile") || warn "Can’t open tmp";

   $notify = 1;

   while (<INPUT>)
      {
      if (/$findstring/ && $notify)
         {
         print "Fixing $file...\n";
         $notify = 0;
         }
      s/$findstring/$replacestring/g;
      print OUTPUT;
      }

   close (OUTPUT);

   #
   # If nothing went wrong (if outfile not empty)
   # move temp file to original and reset the
   # file mode saved above
   #

   if (! -z $outputfile)
      {
      rename ($outputfile,$file);
      chmod ($mode,$file);
      }
   else
      {
      print "Warning: file empty!\n.";
      }
   }
```

Similarly we can search for lines containing a string. Here is the grep program written in perl

```
#!/local/bin/perl
#
# grep as a perl program
#

# Check arguments etc

while (<>)
   {
   print if (/$ARGV[1]/);
   }
```

The operator ‘*/search-string/*’ returns true if the search string is a substring of the default variable `$_`. To search an arbitrary string, we write

```
 .... if (teststring =~ /search-string/);
```

Here *teststring* is searched for occurrances of *search-string* and the result is true if one is found.

In perl you can use regular expressions to search for text patterns. Note however that, like all regular expression dialects, perl has its own conventions. For example the dollar sign does not mean "match the end of line" in perl, instead one uses the ‘`\n`’ symbol. Here is an example program which illustrates the use of regular expressions in perl:

```
#!/local/bin/perl
#
# Test regular expressions in perl
#
# NB - careful with \ $ * symbols etc. Use " quotes since
#      the shell interprets these!
#

open (FILE,"regex_test");

$regex = $ARGV[$#ARGV];

print "Looking for $ARGV[$#ARGV] in file...\n";

while (<FILE>)
```

```
   {
   if (/$regex/)
      {
      print;
      }
   }

#
# Test like this:
#
#  regex '.*'       - prints every line (matches everything)
#  regex '.'        - all lines except those containing only blanks
#                     (. doesn't match ws/white-space)
#  regex '[a-z]'    - matches any line containing lowercase
#  regex '[^a-z]'   - matches any line containg something which is
#                     not lowercase a-z
#  regex '[A-Za-z]' - matches any line containing letters of any kind
#  regex '[0-9]'    - match any line containing numbers
#  regex '#.*'      - line containing a hash symbol followed by anything
#  regex '^#.*'     - line starting with hash symbol (first char)
#  regex ';\n'      - match line ending in a semi-colon
#
```

Try running this program with the test data on the following file which is called `‘regex_test'` in the example program.

```
# A line beginning with a hash symbol

JUST UPPERCASE LETTERS

just lowercase letters

Letters and numbers 123456

123456

A line ending with a semi-colon;

Line with a comment # COMMENT...
```

Example: convert mail to WWW pages

Here is an example program which you could use to automatically turn a mail message of the form

```
From: Newswire
To: Mail2html
Subject: Nothing happened

On the 13th February at kl. 09:30 nothing happened. No footprints
were found leading to the scene of a terrible murder, no evidence
of a struggle .... etc etc
```

into an html-file for the world wide web. The program works by extracting the message body and subject from the mail and writing html-commands around these to make a web page. The subject field of