Introduction to UNIX

Michaela K. Harlander
Physik Department TU München

20. Oktober 1991

This document is an introduction to UNIX for those who have never used UNIX before. It explains the main principles, what you need to create your own files and how to run them through a compiler. A brief overview on networking is given as well as on the X window system.

An alphabetical list of important UNIX commands is provided. You may wish to skim through this list from time to time while working through this introduction.

This document contains a reference card for the `vi` and the `emacs` editor.

Contents

Version 1.0

Comments and suggestions to `Michaela.Harlander@physik.tu-muenchen.de`

Acknowledgement

An introduction to UNIX should be written in a way that a normal user without any ambition to delve deep into computing may understand at least the most essential points. The typical System Administrator, including myself, refuses to remember how he or she felt when first sitting in front of such a crazy UNIX system and trying desparately to force the machine to perform some sensible action.

Therefore, I thank the users at the Physik Department of the Technische Universität München and at the Nuclear Science Division at the Lawrence Berkeley Laboratory. They constantly reminded me of basic problems normal users face when fighting their first battles against UNIX.

The reader probably has already noticed that I'm no native speaker of English. Many thanks go to Wendy Nather and John Sasso for correcting and improving many crude sentences.

In the course of writing a manual, you have to ask the experts for proofreading to fix the errors, misunderstandings etc. Usually, this results in rewriting the complete document. I have been very lucky in this respect – many thanks to Magnus Harlander and Hannes Faestermann who did careful proofreading without telling me that they would have written the whole thing completely different.

Very special thanks go to Bernhard Schneck who heroically worked his way through various versions of this introduction. This resulted in intense discussions on irc and in huge electronic mail messages. He made a lot of very useful corrections, suggestions and remarks which improved this introduction considerably. I also got some remarks of the more, well... hmm..., pedantic sort. Fortunately, we have been separated most of the time by at least the Atlantic Ocean, so none of us took major harm and I'm looking forward to future collaborations. :-)

1 Principles of UNIX

UNIX is a multi–user system. It allows you to share a machine with other users and to run several processes at the same time. You may imagine it to be built like this:

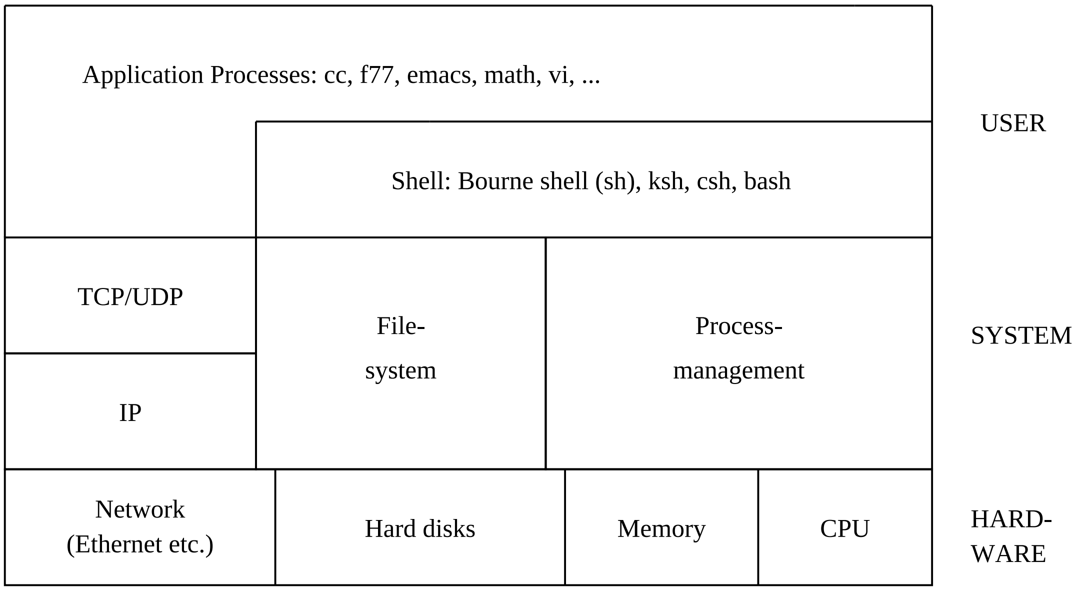

Figure 1: The Structure of UNIX

At the base you find the hardware: the memory, where your processes are kept during execution (although sometimes temporarily stored on a harddisk), the central processing unit (CPU) and eventually a floating point unit (FPU) to do the work, the disks, where your files and all other programs are stored, and the network which connects the machine you are working on to other machines.

The system layer is mainly built out of management systems for the hardware. The process management takes care of your processes and of those of other users so that they do not interfere and share computing resources equally. Only one program at a time can be processed by the CPU, all the others have to wait. You will not see this explicitly since the programs are exchanged in frequent intervals. However, if many processes share the same CPU, each process has to wait longer to be processed by the CPU again and therefore will slow down. All processes with the same priority have to wait the same amount of time. The priority can be changed, however, so some processes get control of the CPU more often than others and run faster.

The filesystem organizes the space available on a hard disk and allows you to see many disks as a homogenous storage medium. If you want to use files that are stored on a harddisk which is attached to a machine other than the one you're working at, the filesystem has to use the network.

To make networking actually work, a general "syntax" is needed for sending data between various machines. This "syntax" is established by network protocols. These come in various levels: While IP is a protocol to merely send packets of data over a cable, TCP and UDP provide further capabilities, based on IP, to make it easier to write application programs. TCP, for example, provides a mechanism to make

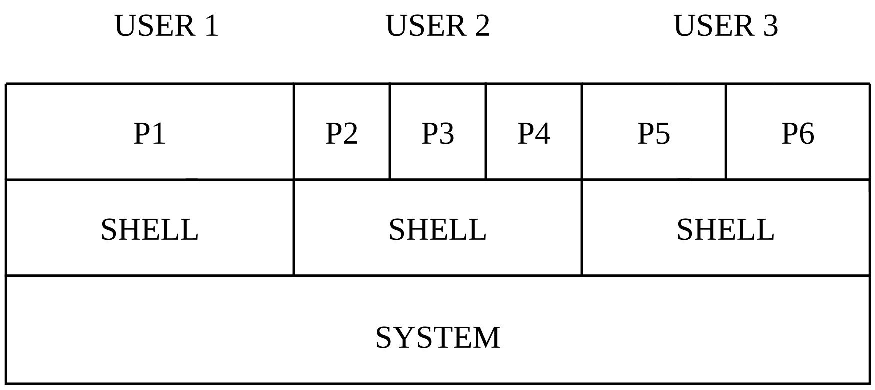

Figure 2: A multiuser system

sure that data actually arrive at the desired destination, even if packets are getting lost. To use network applications, you fortunately do not have to know how these protocols work. Some common network application programs are `rlogin`, allowing you to log on a remote machine (if you've got an account there), `ftp` for file transfer, or mail to deliver electronic mail messages.

In general, however, the main thing you see from your system is the shell which can be best described as a command interpreter. For you as a user there is not much difference between calling a program or using a shell command – except for a very important point: since there are several shells where you can chose one, shell commands differ from shell to shell.

Different flavors of UNIX also employ different commands. Today's UNIX systems come basically in two flavors: those orientated at SYSTEM V, which has been developed by AT&T and those derived from BSD from the University of California in Berkeley.[1] The user interface of these two is different, so some commands available on one system may not be available on the other, or commands have different syntax on various systems. Newer versions of SYSTEM V have been influenced a lot by BSD, however. Especially the latest version, SYSTEM V RELEASE 4 (SVR4) employs a lot of commands originally found in BSD only. Therefore, some of the remarks in this introduction concerning peculiarities of SYSTEM V apply only to old versions like SYSTEM V RELEASE 3 (SVR3). If we state something does not work with SVR3, then this applies also to elder versions of SYSTEM V. In case of doubt, consult your system's manuals.

On the top you find a huge amount of application programs which may be compilers, editors, programs for analytical mathematics etc. These and some shell/system commands are what you probably will use most, besides your own programs.

Fig. 2 shows a scheme of a multiuser and multitasking system. Each user has a shell, which may be different for different users, and can run several processes (tasks), here indicated by P's with a number. They are called children of the (login) shell since they have been started from this shell.

[1]Strictly speaking, only AT&T's product may be called UNIX. In this document, the term UNIX refers to all UNIX-like systems.

Now you know a bit about the system and are ready to actually use it. But before you type any command in UNIX, it is important to know that UNIX is case–sensitive!

2 Getting Started

2.1 Account and Password

Before you can log in, you have to get an account. The usual way to get one is to go to the System Administrator. There you get a loginname which probably is derived from your real name. You will belong to a group. This can be all people of a university institute, all employees of a certain department, all students, etc. The group is important for file access permissions (see section 5.5).

The System Administrator will assign you a login shell. See section 3 for properties of various shells. You can change your login shell using the command `chsh`.

Furthermore, a password has to be provided which should be known only by you and has to be entered during the login procedure to make sure you are the user you claim to be. It is secret! The password is stored on the computer in encrypted form, so no one, not even the System Administrator, can read it. It is very important that you keep it secret!

The password should be at least 6 characters long. The first 8 characters are significant. It should consist of lowercase letters, uppercase letters, numbers and special characters like commas, periods etc. Using control characters can lead to problems. A password consisting of lowercase letters only is not considered to be safe since it is much easier to guess. The same holds for a password consisting only of numbers. Do not use something obvious like your name, date of birth or phonenumber as password nor those of your brother, sister, boy-friend, wife etc. or anything else someone could guess who wants to break into your account! On the other hand, you should not choose a password that is so complicated that you have to write it down to remember. NEVER write a password down! You should also change the password at least every three months or as soon as you suspect that someone might know it. This can be done by the command `passwd` or `yppasswd` if your system is using YP (see section 9).

2.2 Machine Handling

Be careful that you do not switch a machine or its peripherials on or off by accident. A UNIX machine is never switched on or off by anyone except the System Administrator. Simply switching it off may cause big damage to the filesystem. If the screen is dark, hit some keys.[2] Many systems have a screen-saver, which turns the screen dark if there has been no keystroke for several minutes. Hitting a key restores the screen. If this does not work, you may have to switch on the screen. Look carefully to make sure that you really have the correct switch, namely for the screen, not for the machine. Be aware that there are computers which have the machine built into the screen, so the switch for the screen is the same as for the machine. In case of doubt, ask!

[2] `SHIFT` or `CTRL` are good choices since they do not pass input to any application which might be running. If it does not work, you have to try other keys.

2.3 Login Procedure

When you're ready to log in, you'll see something like this on the screen:

```
login:
Password:
```

After you enter a loginname and password, the password is checked and if it is found to be o.k., a shell is started and the environment is set up. What happens exactly depends on various things such as the shell you chose to be your login shell and the kind of terminal you sit in front of.

If you use a workstation or an X terminal, usually several windows are started (for more on the X window system, see section 10). If you use a PC or a terminal connected to the UNIX machine by a serial line, you need a program to emulate a single terminal. A PC with direct access to the net often runs software which enables you to `rlogin` (see below) to a UNIX machine. Some PCs may even run UNIX by themselves.

In any case, the *message of the day* will be displayed, containing up-to-date information on the system. It is important that you read it carefully since things like down times for the system or installation of new software packages will be announced there.

After login you will see a prompt. The prompt is a sequence of characters the shell places at the beginning of a line when it is expecting you to enter a new command. Here, we will use a simple `$` sign as prompt. Note that it could be completely different on your system, for example it can contain the name of the machine or the pathname of the working directory. This is, of course, of no importance for the commands you enter.

Once you're logged in to a machine which is connected to a network, you can also log on other machines where you have an account. This time you have to use the `rlogin` (r representing remote) or the `telnet` program. In contrast to the usual login you have to provide the name of the machine where you want to log in (see section 9.1). Example:

```
$ rlogin heart_of_gold.magrathea.universe
```

If you have a different loginname on this machine, you say

```
$ rlogin heart_of_gold.magrathea.universe -l other_loginname
```

A telnet session looks like this:

```
$ telnet heart_of_gold.magrathea.universe
Trying 7.7.7.7...
Connected to heart_of_gold.magrathea.universe
Escape character is '^]'.
```

```
UNIVERSE UNIX (heart_of_gold)

login: loginname
Password:
```

To log out, you can give the command `logout` or press `^d`, which means "press the Control key and at the same time the `d` key"[3]. Some systems want to see a `^z`.

2.4 Entering Commands

Commands are entered by giving the command name, sometimes followed by options and/or arguments.

Options affect the way a command works. They usually are single letters preceeded by a dash without a space between dash and letter. You saw an example for an option (`-l`) above with the rlogin command. In most cases you are allowed to specify several options by just writing the different options one after another behind the dash. Some programs like compilers, however, require each option to be given separately. In case of doubt, refer to the manpages.

Arguments are usually items which are subject to the command's operation, e.g. if you want to look at a file, you type `more filename`, where `more` is the name of a command to display files page by page, and `filename` is the argument the `more` command shall operate on.

A command line is terminated by hitting the `RETURN` or `ENTER` key.

3 Shells

3.1 Features and Startupfiles

The shell sets your environment, e.g. it sets your prompt, tells the machine where to look for new mail etc. To do this, the shell executes at least one startupfile. Most shells use two of them (see the table). The shell also can have a mechanism called history which remembers the last commands you gave and allows you to repeat these commands identically or modified. Another nice feature is job control, which allows you to stop jobs or to start them in the background, i.e. your terminal does not wait for these jobs to finish and immediately gives you back your prompt, so you can continue to enter commands while the job is processed in the background. In the following table you see which shell has which features.

shell	sh	csh	ksh	bash
1st startupfile	`.profile`	`.login`	`.profile`	`.bash_profile` or `.profile`
2nd startupfile	–	`.cshrc`	`$ENV`	`$ENV`
features	dumb	job control simple history	job control nice history	job control nice history

[3] The Control key can be marked "Ctrl" or, on German keybords, "Strg".

sh is also called Bourne Shell. `$ENV` as second startupfile for ksh and bash means that the name for this startupfile is set by the variable `ENV` in the first startupfile. Often, `.kshrc` is used for ksh and `.bashrc` for bash. The `$` sign in combination with shell variables means that that the value of the variable has to be inserted.

3.2 History and Job Control

The following table gives an overview over history commands.

operation	*csh*	*ksh*	*bash*
repeat last command	`!!`	`r`	`!!`
repeat last command that started with `exp`	`!exp`	`r exp`	`!exp`
show last commands	`history`	`history` or `fc -l`	`history`
repeat command no. `n`	`!n`	`r n`	`!n`
repeat last command, but replace `a` *with* `b`	`^a^b`	`r a=b`	`^a^b`

There are more advanced techniques to modify previous command lines in each of the three shells. For example, ksh and bash allow to search for a string that appeared in a previous command line and displaying this line. Read the manual pages for detailed information.

Job control: On BSD systems, `^z` stops a job. The command `jobs` displays all stopped jobs giving them a number n. Typing `fg %n` gets job `n` back to the foreground. If you just type `fg`, you get back the job you stopped last. Giving the command `bg` after stopping a job puts it in the background. You can do this immediately when starting the job by putting a & at the end of the command. The latter method works also with systems where you cannot stop a job, i.e. for many machines running SVR3 or previous versions of SYSTEM V.

3.3 The `nice` Command

Computationally intensive jobs can slow down a machine immensely for other users even for things that do not require much CPU time like editing. You should use the `nice` command to give your process lower priority. It will run as usual if no other user process is present on the machine but will give user processes like editors priority.

The scale of `nice` values ranges from 0 to 39 in most cases, with 20 as the default user nice value. You cannot give your process a higher priority, i.e. a lower nice value. To nice your program, you use the `nice` command with a nice value followed by your program's name. This starts your program with lower priority. E.g. to increase the `nice` value by 10, use `nice -10 program`. Unfortunately, csh uses the `nice` values with the "wrong" sign, i.e. you have to type `nice +10 program` in csh. `/bin/nice` reads the - as an option while csh interprets it as a sign. To avoid confusion, you can explicitly use `/bin/nice` to get the usual `nice` syntax even in the csh.

Processes with the same `nice` value share the machine resources equally.

4 Getting Help

UNIX systems offer many on–line help tools.

- `apropos expression` is well suited if you do not remember a command completely or simply do not know the command name for a certain feature. The system will look for `expression` in the headlines of its manuals and tell you the context where `expression` shows up.
- `whatis command` gives a short description of `command`.
- `man command` gives you extensive information on `command`. This command invokes the manpages (manual pages).

There is also an information system built into the `emacs` editor. We will come to it when discussing `emacs` (see appendix B.22). Some sites have additional systems for local information.

In case of problems you should also have a look at your system's hardcopy manuals. In most cases, there is a file named "Global Index", which allows you to quickly find the manual you need for your particular problem. Of course, you can also ask other users, the "help desk" or "Programmierberatung" if available, or the System Administrator to help you. Please do the latter only if you have checked all other possibilities or if the problem can only be solved by a System Administrator (e.g. if your account has been disabled).

5 The Filesystem

5.1 Structure

UNIX views a file as a continuous stream of information. You can as well say, it is a sequence of bytes. A byte defines a small unit of information as stored on a computer. Each file has a name. A special file which contains the names of other files is called a directory. As we will see later, the default input and output channels are another kind of special files. Directories are used to structure the files present on your system.

The UNIX file structure is hierarchical. It starts with the root directory / and develops like a tree. Fig. 3 shows a small part of such a tree and as an example, part of the homedirectory of the fictitious user "arthur".

`/home/earth/arthur` is the home directory of the user arthur where he goes after logging in. User arthur belongs to the group `earth`, and in this special case, the directory tree splits up into the groups first before splitting up into the home directories. On most systems, the homedirectory of a user can be addressed by `~loginname` in csh, ksh and bash. Your own homedirectory may be addressed by a single `~`.

There is no limit on the number of subdirectories. It is a good idea to use subdirectories to structure your homedirectory, e.g. to create a directory for C programs and in this directory create a separate subdirectory for each programming project. A directory called `bin` is usually used to hold binaries (executable files).

The following commands are available for dealing with directories:

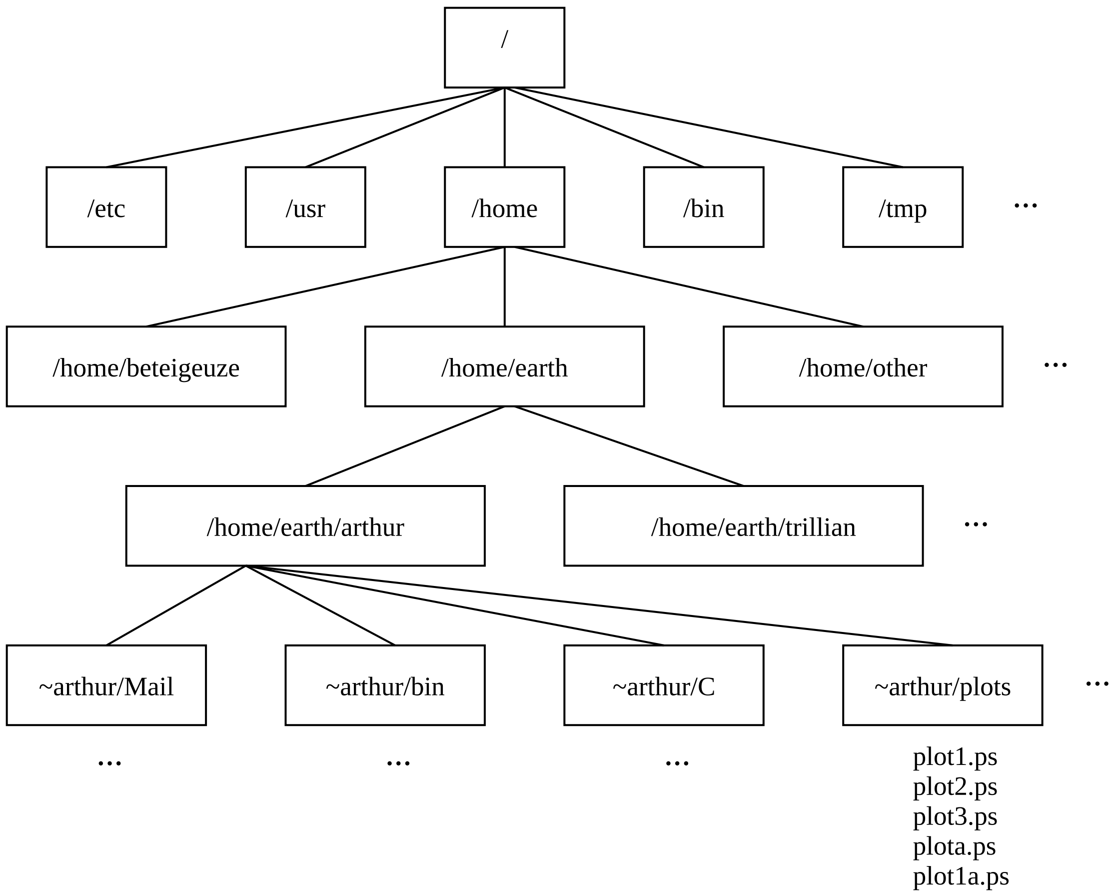

Figure 3: Part of a typical UNIX directory tree

mkdir dirname make a new subdirectory called **dirname**

rmdir dirname remove the (empty!) subdirectory **dirname**

`cd dirname` go to `dirname`

`cd ..` go one directory up in the tree (parent directory)

`cd` go to home directory

`ls` list content of directory

`ls -l` long listing of directory's content

`pwd` print working (current) directory

The working, i.e. the current, directory is represented by a single period, the parent directory by two periods.
Example:

```
$ pwd
/home/earth/arthur
$ mkdir test
$ cd test
$ pwd
/home/earth/arthur/test
$ cd ../C
$ pwd
/home/earth/arthur/C
$ cd
$ pwd
/home/earth/arthur
$
```

5.2 File Names

UNIX does not care much about filenames and allows in principle all characters. However, you're always on the safe side if your filename contains only the following characters:

- uppercase and lowercase letters
- digits
- underscore (_)
- period (.)
- dash (-)

Other characters may be misinterpreted by the shell. If using uppercase letters, keep in mind that UNIX is case sensitive. Filenames may be 253 characters long. Some old SYSTEM V allow only 14 characters.

Contrary to MS-DOS, UNIX does not use extensions (this is something separated from the rest of the filename by a dot). Therefore you are allowed to use as many dots in your filenames as you like. For example, UNIX does not need a special ending (like MS–DOS) to decide whether a file is executable or not. Application programs, however, look sometimes for the last characters of a filename separated by a dot to determine what kind of file it is. Some of these "extensions" that have a special meaning to application programs are

.tex tex source code

.dvi tex code after compiling

.ps postscript files (postscript is a graphics language)

.tar archive file

.Z compressed file

.c C source code

.f FORTRAN source code

There are many more, e.g. LaTeX produces a number of files with various extensions, but you'll soon find out about their meaning. The extensions used by compilers are found in the section dealing with compilers.

Files starting with a period are "invisible", i.e. a `ls` command does not show these files. For example, all startup files begin with a period, since you don't want to see them every time you do a `ls`. To see invisible files in a listing, type `ls -a`.

If you address a file, you can use its absolute or its relative pathname. The absolute pathname contains the full path from the root directory `/` down to the directory where the file resides. `/home/earth/arthur/plots/plot1.ps` is such an absolute pathname. A relative pathname describes the path relative to the working directory. The following relative pathnames all specify the same file:

```
relative pathname          from working directory

 plot1.ps                        ~arthur/plots
 plots/plot1.ps                  ~arthur
 ../plots/plot1.ps               ~arthur/C
```

If we refer to the simple filename without any path, we will sometimes use terms like "base filename".

5.3 Filename Expansion

Sometimes you wish to deal with several files but do not want to type all of them, or you'd like to abbreviate a filename. The shell provides so-called filename expansion to save you typing. The most important mechanisms are:

- the * wildcard character matches any number of characters, even zero. In `~arthur/plots`, `ls *.ps` lists all files which end in `.ps`.
- the ? wildcard character matches any single character. `ls plot?.ps` would list `plot1.ps`, `plot2.ps`, `plot3.ps` and `plota.ps`, but not `plot1a.ps`.

- if you want only some characters to be matched, give the range or a list in brackets. `[ae]` matches an a or an e, `[a-z]` matches any single lowercase letter.
  `ls plot[1-3].ps` lists `plot1.ps`, `plot2.ps` and `plot3.ps`, but not `plota.ps` or `plot1a.ps`.

Especially ksh and bash offer more advanced tools for filename expansion. See the manuals for these shells.

5.4 Renaming and Deleting Files

`mv fname1 fname2` rename file `fname1` to `fname2`; an existing file `fname2` would be destroyed

`mv fname dir` move file `fname` to directory `dir`; the file has the same base filename afterwards, but the pathname is changed. If `dir` already contains a file with the base filename `fname`, this file will be overwritten.

`mv dir1 dir2` move all of directory `dir1` including subdirectories into directory `dir2`. If `dir2` does exist, `dir1` will become a subdirectory of `dir2`. If `dir2` does not exist, it will be created. This does not work on older SYS V systems.

`cp fname1 fname2` copy file `fname1` to `fname2`; an existing file `fname2` would be destroyed

`cp fname1 dir` copy file `fname1` to directory `dir`, keeping the same base filename

`cp -r dir1 dir2` copies recursively all files in `dir1` and its subdirectories to `dir2`, preserving the structure (does not work on older SYS V systems)

`rm fname` remove file `fname`. Contrary to MS-DOS, the `rm` command in UNIX is irrevocable!

5.5 File Access Permissions

Doing an `ls -l plot1.ps` in `~arthur/plots` may give the following output:

```
-rwxr-xr--  1 arthur     3451 Jul 25 13.15 plot1.ps
```

The first dash indicates that you have a simple file. For a directory you would see here a "d". The next 9 characters represent the access permissions on the file. After the number 1 which is a link count[4] you find the owner of the file, then the size in bytes and the date of the last modification. Finally there is the filename.

There are three important things you can do to a file: read it, write to it and execute it. The file access permissions determine who is allowed to do each of these three to your file. The permissions are specified for three different kinds of users: the owner (u), the group (g) and the rest of the world (o).

The first 3 characters of the access permission tell you that the owner may read (r), write to (w) and execute (x) the file. The next 3 characters give the permission

[4] This tells you if there are any links, i.e. alternative filenames for this file. See p. 34.

for the group (`earth` in this example). Group members may read and execute the file, but are not allowed to write to it. If you want to see the group displayed by the `ls` command, use the options `-lg`. The last three characters indicate that the file may only be read, but not written to or executed by all other users not belonging to the group.

Access permissions can be changed with the command `chmod`. This can be done for user (u), group (g), others (o) and all of these three (a). A + or - sign indicates whether to add or to remove a certain permission which is then specified by r, w or x. A `=` sign assings the permission explicitly.

```
chmod u-x plot1.ps
```

makes the file `plot1.ps` non–executable for arthur:

```
$ ls -l plot1.ps
-rw-r-xr--  1 arthur     3451 Jul 25 13.15 plot1.ps
```

There is also a number code for the permissions. The letters r, w, x are assigned the following values:

```
r 4, w 2, x 1
```

The above permission would then be coded by `654`. One could therefore use

```
chmod 654 plot1.ps
```

to perform the above change of the access permissions for `plot1.ps`.

5.6 Backups

In case of a harddisk crash, the files stored on this disk may be lost. Therefore it is necessary to make backup copies of your files from time to time so that you can recover from a disk crash.

On some sites, the System Administration provides backups of the homedirectories as a service. However, to be on the safe side you should do some backups of your own. You should store important files on a separate storage medium, e.g. a floppy disk or a tape.

To do a convenient backup, use the `tar` utility as described in appendix A. This utility is able to write complete directories, including all subdirectories, in an archive file.

You can also compress the resulting `tar`file before moving it to the external storage medium. This saves you space when storing the backup files. A common compressing command is `compress`. See p. 32 for a description of `compress` and an example for doing a backup.

6 Working with the Shell

6.1 Standard Input and Standard Output

Many programs just read input, process it and write some output. Therefore UNIX predefines a standard input channel called stdin. It is just a predefined file, and the content associated with it is usually taken to be the input from your keyboard. If you have several controlling terminals,[5] each of them has a standard input of its own. Similarly, there exists a file called standard output which is commonly identified with the display of the controlling terminal. All output from processes is sent to standard output if nothing else has been specified.

There is a third file called standard error receiving error messages. It also is identified with the terminal, nevertheless it is a file distinct from standard output and therefore may be redirected to another file than standard output. This is useful, if you do not want output of a process and error messages to be mixed up.

6.2 Redirection

Redirection means to assign a different file to standard input, standard output or standard error. Standard input is redirected by a `<` sign.

```
command < infile
```

reads the input for `command` from file `infile`. Standard output is redirected by `>`:

```
command > outfile
```

writes its output to file `outfile`. An existing `outfile` would be overwritten. For appending the output to a file rather then overwriting it, use `>>` instead of `>`.

Standard error can be redirected by `2>` in sh, ksh and bash. If you already redirected the standard output and want to redirect the error messages to the same file, use `2>&1`. Examples:

```
command < infile > outfile 2> errfile
```

takes the input from `infile`, writes results to `outfile` and error messages to `errfile`. The order of redirection of the three standard files is interchangeable.

```
command > outfile 2>&1
```

writes results and error messages to `outfile`. Here the order of redirection is important. Standard output must be redirected before you redirect standard error to the file which replaces standard output.

In csh, you can use `>&` to redirect standard output and standard error to the same file:

```
command >& outfile
```

Redirection of standard error alone is not possible in csh.

[5] This is the screen or window you're operating with. In a window environment each window corresponds to a separate terminal.

6.3 Pipes

Pipes allow you to take a command's standard output as standard input for another command. The pipe symbol is `|`. As an example, assume you want to know whether user "arthur" is logged on the local net. You may use the output of the `rwho` command, which lists all users on the local net, as input for the `grep` routine (`grep` searches for a pattern and displays the lines where it found a match):

```
rwho | grep arthur
```

lists all logins of "arthur" on the local net.

Or you may want a long listing (`ls -l`) of a directory with very many files. To display it page by page, pipe the output of the `ls -l` command into the `more` utility:

```
ls -l | more
```

6.4 Environment Variables

During the login procedure the shell sets up several environment variables. We have listed some of the more important ones below. To display their value, use `echo $VARIABLE`. `echo` copies its arguments to the standard output and the `$` sign references the value of `VARIABLE`.

`TERM` the terminaltype you use, e.g. xterm or vt100 (is used by many programs to find the characteristics and control characters to manipulate your terminal)

`DISPLAY` the display; important if you are running X (see chapter 10.)

`PRINTER` the default printer used by the `lpr` command (see appendix A)

`PATH` list of directories where the shell searches for the commands

`PS1` the string used for the usual (primary) prompt in sh, ksh and bash

`prompt` the string used for the usual (primary) prompt in csh

`HOME` your home directory you enter after you logged in

To change shell variables, use

- in csh and tcsh: `setenv VARIABLE value`
- in sh, ksh and bash: `VARIABLE=value; export VARIABLE`

To add a directory to the path, e.g. if user arthur wants to have his `bin` subdirectory in the path, use:

- in csh: `setenv PATH ${PATH}:$HOME/bin`
- in sh, ksh and bash: `export PATH=${PATH}:$HOME/bin`

The current directory is represented by a dot in the path.

You can put such additions to your path in one of your startupfiles. This saves you typing a pathname when giving a command which resides in a directory listed in `PATH`. If you give a command by just typing a base filename, the shell looks through all these directories one after the other until it finds the command. If it does not find the command, the shell prints the error message `command not found`. To invoke a command which resides in a directory not present in the path, you have to give the full or the relative pathname. To see all directories in the path, type `echo $PATH`.

6.5 Aliases

Sometimes you like to have abbreviations for shell commands you use quite often. You can define aliases for this purpose. It is most convenient for you to put your aliases in one of your startupfiles. In the following example, we define the alias `ll` for the command `ls -l`:

- `alias ll 'ls -l'` in csh
- `alias ll='ls -l'` in ksh and bash

The use of the various forms of quotation marks in the shells is a non-trivial matter. If you get problems, just look at the manpages. The above form should however work in nearly all cases.

6.6 Using the Shell's Special Characters as Normal Characters

Sometimes you want to give arguments on the command line which contain characters that have a special meaning to the shell. You don't want the shell to interpret them, but to take them literally. I.e. you want to escape the shell's interpretation. For example, you look for the occurence of "Smith&Wesson" in a file called `phonelist` where you keep your phonenumbers.

```
grep Smith&Wesson phonelist
```

would not work since the ampersand (`&`) tells the shell that the command ends here and it should be processed in the background. `Wesson phonelist` is interpreted as the next command. To escape the interpretation of the shell, you can chose among the following:

- Put the search pattern in double quotes. Anything in double quotes is interpreted as a string that should be taken literally. References to shell variables are processed, however. Therefore, this method does not work with $ signs in the string. File name expansion (file globbing) is not performed.
- Put the search pattern in single quotes. In contrast to double quotes, references to shell variables are not processed. Also, file globbing is not done.
- Escape the ampersand directly, i.e. precede it by the shell's escape character `\`. The escape character prevents the following character from interpretation by the shell. It has nothing to do with the `ESC` key on your keyboard.

To make the above example work, you could therefore use one of

```
grep "Smith&Wesson" phonelist
grep 'Smith&Wesson' phonelist
grep Smith\&Wesson phonelist
```

6.7 Emergencies

If there is a process doing something you definitely not intended it to do when starting it, you can

- sit and wait—if it is not destructive or using much computing time
- interrupt the process by typing `^c`. This aborts simple commands, others might return to their prompt level or normal mode of operation
- If the program ignores the `^c`, you can try `^\`. This is a stronger request to stop execution, normally also producing a core dump (you'll find a file called `core` in the current directory of the program, which records its state and can be used for debugging)
- kill it from another shell. First do a `ps -ux` command (or `ps -e` on SYS V) to see all your active processes, and look for the process-identification number (`PID`) of the bad guy. Then enter `kill PID`.
- if you have no window system and this process is messing up your screen, you may wish to stop it before killing. After entering `^z` to stop this process, use `kill %`. This command kills the job you stopped last.
- *really* kill it. If the simple kill did not work, try again with `kill -9 PID` or `kill -9 %`.

If your terminal behaves very strangely, not printing commands you enter or giving you funny signs, you have to reset the terminal parameters to some reasonable values. This strange behaviour can result from a `vi` crash. Type `^jstty sane^j`. The `^j` replaces the usual `RETURN`. Some systems want to see `reset` instead of `stty sane`.

6.8 Shell Scripts

The shell can be used as a programming language, too. It provides you with tools like loops, if-statements etc. The commands are different for the various shells, but sh, ksh and bash have a broad subset of commands in common. The exact syntax can be found in the manual pages. If you want to use shell scripts to automate some tasks it is highly recommended to use `sh` since `sh` is available on every UNIX system.

You can use it also to just write some often used commands to a file - everything suitable as input to the shell can be the content of a shell script. Assume, you have such a file, e.g. called `script`. To invoke it, you have to tell the system which shell you want to use. There are two possibilities:

- Invoke the script by explicitly specifying the shell on the command line:

  ```
  sh script
  ```

  You must have read permission for the file to do so.

- Put the following as first line in `script`:

  ```
  #!/bin/sh
  ```

Execute `script` by simply giving its name on the command line. Of course, you must have permission to execute the file. If the first characters of a file are `#!`, the system expects the pathname of an interpreter, e.g. a shell to follow, and uses this interpreter to process the file. On some very old SYSTEM V machines, this method does not work.

We will give some examples here. The first uses the `find` utility. It may be useful to have a look at the `find` manual page to understand the example.

The following script looks in your homedirectory and all subdirectories for files you probably do not need anymore like backup files from emacs (ending with a `~`), `.dvi`, `.aux` or `.log` files from LaTeX runs and core files. Core files are produced if one of your programs fails due to a fatal run time error. They usually are rather big and can consume a lot of diskspace. Doing such a cleanup from time to time helps you to need less diskspace.

```
#!/bin/sh
echo starting cleanup
cd
find . \( -name '*~' -o -name '*.log' -o -name '*.dvi' -o \
       -name '*.aux' -o -name core \) -ok rm {} \;
echo cleanup done!
```

The first `cd` ensures you are in your homedirectory when starting the command. The `find` then starts from the current directory (i.e. the homedirectory), represented by the dot, and looks for files that match one of the names given in the quoted parentheses. The `-name` tells find to look for the name, the `-o` is a logical or. The expressions with wildcards have to be in quotes to prevent interpretation by the shell. The `-ok rm {} \;` tells `find` to remove everything it found, but to ask for confirmation for each file. The two `echo` commands print informational messages.

Here follows a script which makes an archive file of your `tex` and `C` subdirectories and compresses it. This file can then be stored on a floppy disk or tape as a backup file.

```
#!/bin/sh

cd
tar cvf - tex C | compress > backup.tar.Z
echo backup file: backup.tar.Z
```

This kind of shell script is especially useful if you have more complicated commands. The next example summarizes some of the actions a typical user performs in the morning after logging in, surrounded by some cheering remarks. :-)[6] The `#` signs are comment symbols, i.e. the rest of the line is ignored by the shell. This example also contains a `for` loop.

[6] If you don't know what to do with the last three characters: this is a smiley! Just turn your head to the left – see? They are widely used in electronic mail or in netnews to point out irony etc. in case the reader should not have noticed otherwise.

```
#!/bin/sh

for i in 1 2 3      # a for loop in sh, runs from 1 to 3
do                  # belongs to the for loop syntax
echo Good Morning!!!# writes "Good Morning!!!" to the screen
sleep 2             # waits two seconds (to let you wake up)
done                # ends the for loop
echo                # prints an empty line
echo -n Today is    # the -n prevents newline at the end (BSD)
date                # prints date+time
echo
echo Hmmm... who else is on the net at THIS time of day???
rwho | more          # looks for other users on the net
echo
echo What about our machine??
uptime               # displays status of machine
echo
echo Well, let's have a look at the TO_DO list...
cat ~/TO_DO          # many people write the stuff the're ought
                     # to do in a TO_DO list
echo Gee, now make a pretty face and wake up!!
```

7 Editing Files

On most UNIX systems, there are two convenient editors: `vi` and `emacs`. Beside these two, there is also a line editor called `ed`. For non-interactively processing a file, you can use `sed`, `awk` or `perl`.

You can look at a file without changing it by using the `cat` command.

```
cat filename
```

displays the file `filename` in one chunk on the display. To display it page by page, use

```
more filename
```

7.1 vi

`vi` is still the most widely used editor under UNIX, even though it is not as convenient as `emacs` is. Unfortunately, there is no good manpage for `vi`. The reference card included in the appendix may help to overcome this a bit.

You edit a file by typing `vi filename`. If `vi` complains that it does not know which terminal you're on, leave it by entering `:q!` and set the environment variable `TERM` (see previous section). If you have no idea about what your terminal may be, try `vt100`.

`vi` knows three modes: Insert Mode, Command Mode and `ex` Mode (also called Last Line Mode, Bottom Line or : Mode). When `vi` starts, you are in Command Mode, i.e. you may enter commands, but no text. To enter text, give one of the commands `i` insert at current cursor position

`a` append after cursor

`o` open new line below current line

Other commands for adding text can be found in the appendix.

To get back to command mode, which for example is necessary to move the cursor, press the `ESC` key. If you do not remember which mode you're in, press `ESC`: if you were in Insert Mode, you're now in Command Mode, if you were in Command Mode, `vi` beeps at you and leaves you in Command Mode.

For more complicated commands, you have to use the Last Line Mode. To get there, type a colon in Command Mode. Everything up to the next `RETURN` is then taken as a command. Commands in Last Line Mode are different from those in command mode. Here and in the `vi` reference card, all commands in Last Line Mode are preceded by the colon which enters this mode.

`vi` is case sensitive. Since you do not see the commands you type in command mode, check for the `CAPS LOCK` key in case `vi` refuses to understand you!

If you see a ˜ at the beginning of a line this means that this line does not belong to the file. This is used to mark the end of a file.

You can quit vi by typing `ZZ` in command mode. This stores all changes you made to the file. If you want to quit `vi` without storing the changes, type `:q!`.

The most important commands for `vi` are collected in appendix C.

7.2 emacs

`Emacs` is a non–proprietary software as provided by the GNU project. It is more then just an editor. It is a programming environment. It allows you to read your mail and it has a convenient information system. Using it, you will soon discover that it is able to do even more than that.

To start `emacs`, simply type `emacs`. If you are using it for the first time, type `^h t` to enter a tutorial. An overview over the most important commands is given by the reference card in appendix B.

Read in a file by typing `^x^f` and give the filename. `Emacs` looks for the extension of the file and for some it will start a special mode. For example, the C mode indents C source code properly while you are typing, the TeX mode allows to check for `$` signs and braces etc. For information about the current mode, type `^h m`. To save the file, type `^x^s`.

The info system can be invoked by `^h i`. There you find a description of the info system itself and all other available topics, including emacs.

To leave `emacs`, type `^x^c`. This will ask you if you want to save the changes for each file you were editing since last saving it. `Emacs` is continuely doing backup copies of the files you're editing. You normally do not see them since they are removed each time you save `filename`. They are, however, very useful in case of a system crash. They are named `#filename#` where `filename` is the original name of the file you were editing. If addressing one of the backup files with shell commands, you must escape the `#` signs since they are comment symbols to the shell. `emacs` also keeps a copy of the last version of the file you saved before the current one. This file is named `filename~`.

8 Compilers

The most common compilers on UNIX systems are:

`cc` the C compiler; often available in an ANSI and a non-ANSI version (ANSI C is a standardized C)

`gcc` the GNU C compiler, which is ANSI compliant. Look for the info entry in emacs.

`CC` C++ compiler (there can be other names, too)

`g++` the GNU C++ compiler, see the info entry in emacs.

`f77` the FORTRAN compiler, often a very much extended FORTRAN77

`pc` the pascal compiler - the flavor of pascal it understands depends completely on your system, see the manual for details

The compilation process consists of the following steps:

1. Preprocessing. All comments are removed. Preprocessor directives are inserted when using a compiler which provides a preprocessing mechanism, e.g. a C compiler.
2. Compiling.
3. Optimizing if specified when compiler is called. Compiling and optimizing result in an assembly language file.
4. Assembling. Translates the assembly language file into an object file, i.e. into a language the machine can really handle.
5. Loading. All object files and the appropriate libraries are linked together to an executable.

The following extensions are commonly recognized by the compilers:

.c C source code file

.cc C++ source code file (sometimes also .C)

.f FORTRAN source code file (sometimes also .f77 or .for)

.p pascal source code file (sometimes also .pas)

.s assembler file

.o compiled object code file (not yet linked)

.i C file after preprocessor run (if compiler is called with -P option)

The following options are known to most compilers. You have to give compiler options each separately preceded by a dash. For the arrangement of options and arguments you have to examine the manual pages. Typical examples are given below.

`-O` produces optimized output; the executable runs much faster. Often you may choose among several optimization levels.

`-g` creates symbolic information to be used by source code debuggers (see later in this section)

`-c` does only compile, but not link. The result is a file with extension .o

`-o outname` will name the executable to `outname` instead of `a.out` which is the default

`-l library` links the object file with the specified library. For example, if you are using mathematical functions, you have to link with the math library, which is typically called `libm.a`. Since all libraries have the form `lib`*x*`.a` you only have to specify the *x* part of the library name immediately after the `-l` without a space in between. Linking with the math library is done therefore by `-lm` (see the example below).

To compile the file `first_try.c` with `cc` having optimization turned on and not wishing for linking, you would enter:

```
cc -O -c first_try.c
```

To compile the file `expo.c` using `gcc`, with creation of symbol tables for debugging, linking with the math library and calling the final executable `expo`, you do

```
gcc -g -o expo expo.c -lm
```

Often, a program does not work as expected when started for the first time. For pure syntactical errors in a C program, try running the syntaxchecker `lint`.

Using a source code debugger allows you to go step by step through your program, examine the value of parameters etc. while simultaneously looking at the source code which is executed. Unfortunately, many debuggers support only C. There may be problems with other languages. On many UNIX systems, the debugger `dbx` should be available. If not, look for `sdb`. Very good debuggers are `gdb` and `ups`.

To use these debuggers, your program must have been compiled with the `-g` option. On most compilers, it is not possible to have `-g` and `-O` at the same time.

To make the compilation process more convenient there exists the `make` utility (see also appendix A). It allows to automize the compilation process which is especially useful for large programming projects.

To keep track of various versions of a program there are utilities like `RCS` or `SCCS`. They allow you to retrieve earlier versions of your program without the need to store complete files. These utilities are not available on all systems.

9 Networking

In this introduction the net has been mentioned several times already: you can log on a remote machine, you can `talk` to a person on another machine etc.

There are several kinds of network. The differences between these networks are not primarily based on the physical cables they use but on the protocol spoken on this cable. There is e.g. BITNET, HEPNET or SPAN (DECNET networks), the Internet with the TCP/IP protocol suite, and the OSI protocol family which is supposed to be the standard protocol one day. Today, TCP/IP is most widespread. It is built into most UNIX systems. The following is restricted ourselves to application programs built on top of the TCP/IP protocol.

9.1 Domain Names

Network application programs require that you name a machine. Each machine has a unique, so-called “domain name”. These names typically look like

```
machine.subdomain1....subdomainn.top-level_domain
```

It consists of the machine’s name and several subdomains, separated by a dot. The last domain is called top-level domain. Each country participating in the domain name system is assigned such a top–level domain, usually the ISO 2–letter country code. As an exception, the USA has seven top–level domains:

EDU educational institutions

COM commercial organisations

ORG non–commercial organizations

GOV governmental institutions, e.g. National Laboratories

MIL military institutions

NET major network support centers

ARPA hardly used anymore, originally institutions specific to the Internet

Some of the country codes are

AU	Australia	AT	Austria
CH	Switzerland	DE	Germany
DK	Denmark	FI	Finland
FR	France	IL	Israel
IT	Italy	JP	Japan
KR	South Korea	MX	Mexico
NL	Netherlands	NO	Norway
SE	Sweden	UK	Great Britain

For the local network at your site, aliases may be defined in order to safe you typing the long domain names. E.g., the machine

```
heart_of_gold.magrathea.universe
```

could be addressable as `heart_of_gold` in its local network.

9.2 Remote Login

The two possibilities for remote login are `telnet` and `rlogin`, as already explained in section 2.3. Since their use has been described there, we will concentrate here on the main differences between these two.

- `rlogin` works only with UNIX machines. To connect to non-UNIX machines, you have to use `telnet`.
- To connect to very remote machines, use `telnet`. If you are connected to a remote computer and enter a command, every single character is sent over the network wrapped in an data packet of its own. It would be much more economic if the whole comand line would be sent in one such packet. This gets essential if the packets have to travel very far, using lots of network resources. In `telnet`, just type telnet's escape character (most often it's `^]`) and the command `mode line`. This will process the command line at your local machine, and send the whole line over the net after you pressed the `RETURN` key. This does, however, not work for editors. To swich back, type `^] mode character`.
- `rlogin` allows remote login without supplying a password by using a file `.rhosts` on the destination machine. In this file, machines can be listed from where a `rlogin` may be accepted without prompting for a password. For security reasons, it is strongly recommended NOT to use this feature: if anyone succeeded in cracking into one of your accounts, she/he can enter all accounts which allow your `rlogin` from the cracked account without asking for a password. A `.rhosts` file can be tolerated if limited to a local net where you have only one password for several machines. In any case, your `.rhosts` for security reasons MUST have the access permission `400`, i.e. only readable for you.
- `telnet` allows you to do more then just remote login. Refer to the manpage.

9.3 File Transfer

There are two utilities for file transfer: `rcp` and `ftp`. `rcp` is not recommended since it needs a `.rhosts` file. `rcp` is not suited for large file transfers.

A ftp (file transfer protocol) session start for user `arthur` connecting to `heart_of_gold.magrathea.universe` looks like this:

```
$ ftp heart_of_gold.magrathea.universe
Connected to heart_of_gold.magrathea.universe
220 heart_of_gold FTP server (UNIVERSE OS 77.1) ready.
Name (heart_of_gold.magrathea.universe:arthur): arthur
331 Password required for arthur.
Password:
230 User arthur logged in.
ftp>
```

You give the machine name as argument to the ftp command. You have to enter loginname and password to get access to the remote machine. Ftp then expects ftp commands. Entering a `?` gives you a list of available commands. Some important commands are `ls` or `dir` list directory on remote machine

`cd dirname` change to directory `dirname` on remote machine

`lcd dirname` change to directory `dirname` on local machine

`binary` transmit the files in binary mode. This is very useful for transfering large files since it is much faster. To switch back to the default value, enter `ascii`. Binary mode should be used for all file transfers. However, you can not use it if

- at least one of the machines is a non–UNIX machine
- transfering ASCII text files between different architectures

`hash` prints hash marks (`#`) during file transfer so you can see how fast the transfer is going on. The number of transfered bytes a single hash mark corresponds to depends on the ftp server and is displayed when switching hash mark printing on.

`get fname` get file `fname` from remote machine to local machine

`put fname` put file `fname` from local machine to remote machine

`mget filelist` get files in `filelist` from remote machine.

`mput filelist` put files in `filelist` to remote machine.

`prompt` switch interactive mode on/off. This is useful for the `mget` and `mput` commands which prompt you for confirmation for each file's transfer.

`quit` close connection and end ftp session.

You can not transfer binary numbers and floating point numbers between machines with a different hardware architecture (e.g. from a DECstation to a SUN SPARCstation).[7]

Some sites provide *anonymous ftp*. These sites have archives where they store free software. You can use anonymous ftp without having an account for the machine where you want to get software from. Just give `anonymous` for the username and your loginname (!) as password. NEVER give your password when using anonymous ftp!! Information about anonymous-ftp servers is most often obtained from netnews (see below.).

However, before you try to get files from very far away, look at archive sites that are close to your site to avoid unneccessary network traffic.

9.4 Electronic Mail

The network enables you to receive and send electronic mail messages. There are many programs you can use to handle your mail. Simple programs are `mail` or `mailx` which are supplied with almost every UNIX system. Built into `emacs` is `rmail`. You can start the latter by typing `ESC x rmail` in `emacs`. Type `^h m` to get a description of the mail mode you entered. Another very convenient mail handler is `mh`, which

[7] Please note the difference between e.g. a floating point number stored using the machine's floating point representation as can be done by C's `fwrite` command or a floating point number stored as an ASCII string.

can be available from the shell, with an X interface or within `emacs`. There might be other mail programs at your site.

How to read mail depends completely on the mail program you use, therefore we will not comment on that. There are, however, some things you should know about sending mail.

First, you have to know the mail address of the person you want to send mail to. I assume for the moment that this person has access to the internet. Mail addresses have the following form:

```
name@domainname
```

`name` is the name of the mailbox. It is common in UNIX systems to use the loginname of a user as name for the mailbox. Some systems allow to use the full name. There may be other names, too, for example names for a mailing list where the mail sent to this mailbox is distributed to several people.

`domainname` is the domain name of the host where the mail should be received. This is not necessarily the machine where the person you want send mail to usually works. You have to know the correct hostname. The `domainname` can also be an alias defined by the site you want to send mail to in order to have shorter and easy to remember mail addresses (see the `cc:` field in the mailheader below for an example).

A typical mailheader looks like this—with slight modifications depending on your mailprogram.

```
To: juser@foo.bar.edu
cc: arthur@magrathea.universe
Bcc: my_loginname
Subject: demonstration
--------
Here comes the mail body...
```

`To:` is the person to receive the mail, `cc:` are addresses to receive a carbon copy. These addresses will appear in the mail header of the mail each receiver gets. The `Bcc:` field contains addresses to receive "blind carbon copies", i.e. these addresses will not appear in the mail header the recipients in the `to:` or `cc:` fields get. This is most often used to get yourself a copy of the mail. Since for mail on your local net you usually do not need to give a hostname in the mail–address, just insert your loginname in the `Bcc:` field for your copy. In the default mail header, you will probably find the `cc:` or the `Bcc:` field, not both of them, nevertheless you can use both. All of these fields can have multiple entries, separated by commas.

If you sent your mail you will not be informed upon correct arrival of your message. The fact that you got your carbon copy right does not imply that the other recipients got the mail. You will be informed if the delivery fails, however. If you are not informed about a delivery failure within several days, you can be pretty sure that everything worked fine.

If your mail can not be delivered, you get a note with your mail returned. In the note you will find the reason for failure. Some common found reasons are:

`unknown user` The user part of the address, i.e. the name, did not match any possible mailbox at the host/site you were mailing to. Check the name for spelling errors!

`unknown host` The host you specified could not be found.

`host has been down for ...` This means that the host could be found but is currently not responding to mail transfer requests.

If you want to send mail to person not having access to the internet, but having a bitnet address, use the bitnet address with a `.bitnet` attached to it.

Some mail addresses contain `%` signs like in

```
user%host1@host2.domain
```

This means only that the mail has to use a gateway (`host2`). This information is of no importance for you.

You may run over two other forms of mail addresses which are likely to cause more trouble:

- ```
  hostn! ... host2!host1!user
  ```

  is a UUCP address. You may try to translate it in

  ```
  user@host1.uucp
  ```

  If this does not work, you have to specify at least one gateway in the mailaddress. You can try something like

  ```
  user%host1%host2 ...hostn-1@hostn.uucp
  ```

  or

  ```
  hostn-1!hostn-2!....host2!host1!user@hostn.uucp
  ```

  or ask your local mail guru.

- ```
  host::user
  ```

  is a DECnet address. These addresses are very hard to transform to a domain address, if at all. Try to get another mail address for this person!

9.5 Netnews

There exists a worldwide information system called `netnews` or `news`. This system has more then a thousand topical newsgroups ranging from pure technical computer discussions over political disput to recreation information. How many newsgroups you actually can access is site dependent.

To read news, you have to use a newsreader. There are plenty of them, and each works differently. Check at your site which newsreader is available.

But beware! The first time you'll read news, you will be subscribed to all newsgroups with about 120.000 articles waiting for you! Of course you can unsubscribe to newsgroups.

9.6 The Local Net

The net is not only important for remote login or mail. If your site has several computers, it probably uses a network to make working with all these machines more convenient for you. This is mainly achieved by the following:

- Network Information System (NIS), formerly called Yellow Pages (YP)[8] keeps information about the user as found in the `/etc/passwd` file on a central YP server machine in the network. The files are available to all other machines. As a consequence, your password and shell are the same on all machines which are in the same YP domain, i.e. which get their information from the same YP server. If you change your password or shell on one machine, no matter on which one, this change will be stored on the YP server and is therefore known to all machines in the YP domain.
- The Network File System (NFS) allows to share diskspace over the net. This is especially important for your homedirectory. NFS makes it possible that your homedirectory is the same on all machines on the local network. Physically, your homedirectory is stored on a harddisk attached to one machine. Other machines can `mount` the stuff on this disk and it is then available on these machines as if it came from a local disk. The machine which `exports` the files from its local harddisk is called a fileserver. This is also used for application programs since it is much easier to maintain 1 copy of a program that is shared over the net then 20 local copies. Often, only very few software is held on local disks while the rest is stored on disks attached to one or a few machines.
- Printing can also be a network-wide service. This allows you to address any printer you like in the local network, no matter to which machine this printer is physically connected.

A local network does not have to look like sketched above. However, you will find it convenient if it does.

10 The X Window System

The X Window system is a freely available window system developed at MIT. It runs on various platforms and allows you to get windows from other machines over the net. The portability and the network capabilities of X make it superior to other window systems like e.g. Sunview. There are commercial products (Openlook and Motif) which heavily rely on X11 and are similar to use.

If you log on a machine which can run X, it is sometimes already running und you get some windows after the login procedure completed, or you have to start it directly by typing `xinit` or a similar command.
There are many applications available under X:

- the `xterm` terminal emulator (these are the nice windows where you can enter commands); a simple `xterm` is started by `xterm &`.

[8] NIS/YP and NFS have been developed by SUN Microsystems and are licensed by many other manufacturers. These programs are today's standard.

- several window managers; they do the housekeeping and decoration of your `xterms` and other windows.
- several nice utilities, e.g. `xclock` showing you a clock on your screen, `xbiff` to inform you visually about new mail etc.
- graphics programs. There are many of them, and each site has different ones. Some common programs are `xfig`, `xvgr` and `khoros`.
- X is also a graphics programming tool you can use to visualize the results of your calculations.

Many things concerning X like what window manager you use or which graphics programs are available are site specific, so they will not be discussed here. In the following sections, some basics concerning X are pointed out, so you'll be able to fight with your local system – whatever jungle that may be!

10.1 The Role of the Mouse

Sitting in front of an X window display with several windows open, you see that moving the mouse moves the cursor over the screen. If it is inside a window, this window can be marked active, e.g. by a change of color in its titlebar, and you can enter commands. Some window managers force you to activate the window by pressing a mouse button while the cursor is in this window. Nothing happens if you try to enter command while no window is active.

The pure background where all your windows are placed on is called root window.

The effect of pushing one of the three mouse buttons depends on the context in which this button is pressed. The context consists of

- the background where the cursor was when the button was pressed: was it on the root window, on a window body, on a titlebar?
- the keys on the keybord which were simultaneously pressed with the mouse button.

Here's a list of common contexts and what they can affect. The resulting actions mentioned in this list can only be examples since in most cases they depend on the window manager and on the configuration at your site.

root window pull-down menues e.g. for logins and window operations. This is completely site dependent.

window body + meta or alt key this can be for windows which are lying on top of one another: to bring them on top or on bottom of the "window stack". This depends on the window manager.

window body + control key menues with some terminal and window options like enabling a scrollbar or changing a font. This is configuration and site independent.

titlebar of window move a window, bring it on top or on bottom of a window stack. This depends on the window manager.

resize marker of titlebar allows to resize a window. Depends on the window manager, but is commonly used.

icon marker of titlebar makes a very small box (icon) out of the window and puts it in some corner of your screen. Depends on the window manager, but is commonly used.

icon restore the window which has been iconized generating this icon.

Test the effect of pushing the mouse buttons in the different contexts!

10.2 Startupfiles

Similar to the shell, the X Window System has startupfiles. They control things like default color and font of a `xterm`, which applications should be started automatically during login, where shall system messages (console output) appear, etc. Unfortunately, there is no general naming convention for the X startupfiles. These names are completely site specific.

10.3 The Window Manager

This thing is—besides you—responsible for the mess on your screen. It keeps usually a bar at the side where all your X-applications are listed. It may decorate the `xterm` with a titlebar, an icon marker and a resize marker. It often provides several pulldown menues accessible by pressing the mouse buttons on the root window.

One of the window manager's tasks is to place and to move windows. If a window appears, the window manager usually displays a grid attached to the mouse. Move it where you want it to be and press a mouse button to activate the window

You will have a startupfile for the window manager, too. For example, `twm` has `.twmrc` as a startupfile. There you will find a description of the menues available for different contexts.

10.4 xterm

The simplest way to start an `xterm` is to type `xterm &`. Do not forget to start it in the background, otherwise the terminal where you started the new `xterm` will be waiting for this `xterm` to finish before giving you back the prompt.

There are several options which control the way the `xterm` looks and behaves. Here's a list:

`-fn font` the window will have the font specified. Unfortunately there is no convenient way to find out what fonts are available. You can try the commands `xlsfonts`, `xfd fontname` and `xfontsel`. Some common ones are 6x10 (very small), 6x13, 7x14, 10x20 (large).

`-g geometry` this gives the size and location of the winow. `Geometry` is a string of the form "columns x rows +xoffset +yoffset" where xoffset is measured from the left margin of the screen and yoffset is measured from the top. Replacing the + sign by a - measures from right and from the bottom, respectively. xoffset and yoffset are measured in pixels. You may give only columns and rows or only the location. There may be no spaces.

`-bg color` sets the background color. Colors can be addressed by color names. They are stored in a file called `rgb.txt` which resides somewhere in the directory tree where the X window stuff is.

`-fg color` sets the foreground color.

`-n name` The name of the window when iconized

`-T title` The title of the window appearing in the titlebar

`-e command` `command` is executed when the window has been activated. This must be tha last option.

```
xterm -fn 7x14 -g 80x60 -n man -T man -e man xterm &
```

starts a long window with a somewhat bigger font that will be named "man". It will start with the manual page for `xterm`. Some window managers ignore part of the options.

10.5 Selected X Utilities

The following utilities have some options in common. With `-bg` and `-fg` you can chose background and foreground color. The `-geometry` option specifies size and location. Contrary to `xterm`, the size has to be specified in pixels in most cases.

xbiff small mailbox that changes color and beeps if mail arrives

xload shows the load on your machine graphically

xclock an "analog" clock. With the -d option it is a digital clock, also displaying the date

oclock "analog", round clock

xcalc scientific desktop calculator

xman a nice X version of the usual man command

10.6 Using X on the Net

Think of a `xterm` where you `rlogin` to another machine. You want now to have a X-application started on the remote machine but displayed on the local machine. This can be useful if there's a special program only available on the remote machine, for example. There are two things you have to do:

- allow the remote machine to open windows at your screen. To do so, give the command

  ```
  xhost remote_machine_name
  ```

  at your local machine.

- tell the remote machine where to display the X application by setting the `DISPLAY` variable to the local machines' display. The display name is

  ```
  machine_name:0
  ```

  if only one screen is attached to the machine. Else use the number of the terminal you're using instead of the `0` in the last place. E.g., in sh, ksh and bash you would give the following command at the remote machine:

  ```
  export DISPLAY=local_machine_name:0
  ```

If done so, simply give the command for the X application on the remote machine.

There are some dumb programs under X which complain if you want to execute them on your local machine. Check if `DISPLAY` is correctly set on your machine – this can be also necessary for non-dumb programms – and give the `xhost` command with your local machine name. I know, it's ridiculous, but what can I do?

A Important UNIX Commands

In alphabetical order, we will list here some important UNIX commands together with often needed options. Of course, this list does not replace the manpages nor is it complete. It just introduces you to some important commands and tells you about others without exactly explaining how to use them. The descriptions hold for both BSD systems and SYSTEM V if not stated otherwise.

A.1 awk

`awk` is a utility to search for patterns and process them. It is more powerful then `sed`, but not as versatile as `perl` is.

A.2 cat

```
cat filename
```

displays the file `filename`. The output can also be redirected to another file:

```
cat file1 > file2
```

writes the content of `file1` to `file2`.

```
cat file1 file2 file3 > file4
```

catenates the first three files and redirects the output to `file4`.
But beware! `cat file1 file2 > file1` destroys `file1` since it is first opened for writing, i.e. truncated to zero length, before the `cat` command is performed.

A.3 cc

is your systems C compiler. The simplest way to use it is:

```
cc file.c
```

which compiles `file.c` and results in an executable `a.out`. For options, see section 8.

A.4 cd

```
cd dirname
```

changes the working directory to `dirname`. See also section 5.1.

A.5 chmod

changes the access permissions of a file. See section 5.5.

A.6 compress, uncompress and zcat

These commands are to compress or expand files.

```
compress filename
uncompress filename.Z
zcat filename.Z
```

`compress` reduces the size of `filename` and places the compressed file in `filename.Z` if possible.

`uncompress` restores the compressed file to its original form. The compressed file is deleted.

`zcat` is the same as `uncompress`, but leaves the compressed file intact. The output is written to standard output.

To make a compressed archive of your C subdirectory, you could use:

```
tar cvf - C | compress > C.tar.Z
```

This results in a file C.tar.Z. The - sign in the `tar` command tells `tar` to write its output to standard output which then is piped as input to `compress`. Since `compress` gets its input from standard input, it does not know how to replace this with a `.Z` file. Therefore, explicit redirection is necessary. To restore the content of the archive file, use

```
zcat C.tar.Z | tar xvf -
```

The `-` sign at the end tells the `tar` command that it shall take its input from standard input, which is redirected to come from the `zcat` command.

A.7 cp

```
cp source-file destination-file
cp file-list directory
```

`cp` copies files and directories. See section 5.4.

A.8 date

displays time and date.

A.9 diff

```
diff file1 file2
```

displays the differences between `file1` and `file2`.

A.10 echo

copies its arguments to the standard output. Can be used to print the value of a shell variable (referenced by the $ sign).

```
$ echo a
a
$ a=b
$ echo $a
b
$
```

The `a=b` command is not valid in `csh`. You have to use `set a=b` instead.

A.11 file

```
file file-list
```

classifies the files given in file-list according to their contents, e.g. ascii text, executable, C source code, etc.

A.12 find

```
find directory-list expression
```

selects files from directory-list which match `expression`. `Expression` can also contain commands to be applied to the file. `Find` has a fairly complicated syntax and may be time-consuming. NEVER start a `find` from a directory on top of the complete directory tree, like `/` or `/home` - it would look through all the directories beneath `/` or `/home` to find your file!! This may take several hours to complete and make the machine almost unusable, depending on file system and organisation.

A.13 finger

```
finger name@host
```

`finger` displays information about users that are currently logged on a machine. If only a loginname is specified as argument, `finger` looks at the machine you're working with (sometimes also on the local net). You can also finger persons on remote hosts.

The option `-l` gives longer information.

A.14 grep

```
grep pattern file-list
```

searches for `pattern` in files from the file-list, which may contain directories, in which case all files of that directory are being searched. Given you have a list with phonenumbers called phonelist and want to look for the entry "Peter", enter

```
grep -i "Peter" phonelist
```

The `-i` option makes the search case-insensitive.

A.15 head

```
head  -n filename
```

displays the first `n` lines of file `filename`. The default for `n` is 10.

A.16 kill

```
kill PID-list
```

terminates the processes identified by `PID-list`, where `PID` stands for process-ID. The `PID` can be determined by running the `ps` utility. You can only kill processes you own. If the `kill` command does not terminate your process, try `kill -9`.

In shells which support job control, you may use

```
kill %jobnumber
```

where `jobnumber` is the number displayed by the `jobs` command (see section 3.2). `kill %` kills the job you stopped last.

A.17 ln

```
ln file link
```

establishes a so–called hard link to an existing file, i.e. you can access the file by two names: the original filename and the just created name of the link. Both names have the same status. If you remove one of the two names, the file is still accessible by the other one. On newer systems, you can create a symbolic link by giving the `-s` option. This link has not the same status as a filename. It only creates a small file which contains the path of the original file. Therefore, if you remove the original file, the symbolic link is only a pointer to a no more existing file. Contrary to the hard link, a symbolic link can be made to a directory.

A.18 lpr

```
lpr -Ppname file-list
```

sends the files of file-list to the printer named `pname`. Inform yourself about the printernames at your site. Typical names are "lp" for a lineprinter and "ps" for a laserprinter able to understand postscript. You can set the `PRINTER` environment variable (see section 6.4) to a printer name which then is the default printer, i.e. if you use `lpr` without the `-P` option, the printer specified in `PRINTER` will be used.

Never send a postscript file to a lineprinter, it will just print all the postscript commands but not the text or figure you actually want to see. If it happens by accident, immediately take the job out of the printerqueue using `lpq` and `lprm` (see below) since it would waste a huge lot of paper!

You can look at the jobs waiting to be printed at a particular printer by

```
lpq -Ppname
```

This shows you also the job–number for each printing job. If you want to remove a job from the printqueue, use

```
lprm -Ppname job-number
```

where job–number is the job–number displayed by `lpq`. You can only remove your own print–jobs.

The appropriate SYSTEM V commands are `lp`, `lpstat` and `cancel`.

A.19 ls

```
ls file-list
```

displays information about the files in file-list. File-list may also contain directories in which case all files from that directory are listed. If the file-list is empty, the files in the working directory are listed. The following options are most useful:

`-a` displays also files starting with a period

`-l` displays long information about files, including access permissions

`-F` display a / after each directory, an asterisk after an executable and, on BSD systems, an @ after symbolic links.

A.20 mail and mailx

`mail` (BSD) and `mailx` (SYSTEM V) are simple utilities for reading and sending electronic mail. They provide very limited possibilities for editing the mail you write. We recommend to use rather one of the mail systems embedded in `emacs`. See, e.g. the description of rmail in the `emacs` reference card of appendix B.

A.21 make

is a tool to make the compilation process more convenient. It allows to recompile automatically only those files that changed after the last compilation, and those which depend on these changed files. The command `make` executes a file called `Makefile` or `makefile` containing instructions for the compilation process. For syntax and detailed description, look at the manpage. If the GNU version of `make` is available at your site, use the appropriate info entry in `emacs` which is a nice `make` tutorial.

A.22 man

```
man command
```

displays the manual page for `command`. Using the `-k` option with a keyword as argument, you get topical information about available manual pages for this keyword.

For further information on getting help, refer to section 4 and to appendix B.22.

A.23 mkdir

```
mkdir directory-list
```

creates the directories given in directory-list.

A.24 more

```
more file-list
```

displays the files in file-list page by page. On some old SYSTEM V machines, you have to use `pg` instead.

A.25 mv

```
mv existing-file new-file
```

renames a file. See section 5.4.

A.26 nice

lowers the priority of a process in order not to block a machine for other processes. See section 3.3.

A.27 nohup

```
nohup command
```

ensures that the process started by command is not terminated if you log out. If you do not specify output files, both the standard output and error are redirected to a file named `nohup.out`.

If you are using csh, starting a process in the background already ensures that the process is not terminated by logout. Therefore you no not need to use `nohup`. This is not the case for tcsh.

A.28 ps

displays information about active processes. Without any options, you only get information on your processes which are started from the terminal where you entered the `ps` command. Remember that in a window system, each window corresponds to a separate terminal. All of your processes can be obtained by using

```
ps -ux
```

To see other users' processes, too, enter

```
ps -aux
```

If you're interested only in running processes, use the `r` option. The output of a `ps` command consists of several columns. The most interesting are:

`USER` process' owner

`PID` process–ID

`%CPU` the percentage of CPU time the process is using

`%MEM` the percentage of memory the process is using

`SZ` memory size of the process

`RSS` current resident memory size of the process

`TIME` the CPU time in seconds the process used up to now

`COMMAND` the command which started the process. If its truncated and you want to see more of it, add a `w` once or multiple to the `ps` options.

The options are quite different on SYSTEM V and so is the output. The analog to the BSD "aux" option is "ef". By "-u" followed by a username you get all processes belonging to that user.

A.29 rm

```
rm file-list
```

removes the files given in file–list. The effect of this command can not be undone!

A.30 rmdir

```
rmdir directory-list
```

removes the empty directories given in directory–list. Directories not being empty will not be removed.

A.31 ruptime and rup

These commands show the status of the machines attached to your local network.

A.32 rwho and rusers

These commands show who is logged on the local network.

The `rusers` command takes more time, but `rwho` is disabled on some sites since it causes a lot of network traffic.

For options, see the manpages.

A.33 sed

is a non–interactive editor. See the manual page for more information.

A.34 sort

sorts files in ascii sequence or other sequences controlled by options (e.g. -d is dictionary order). The sortfield in the file can be chosen. See the manpage.

A.35 tail

```
tail filename
```

displays the last part of file `filename`

A.36 talk

```
talk loginname@hostname
```

is asking the user `loginname@hostname` to talk to you over the net. If the user is on the same machine as you are, you only need to specify the loginname.

To end a `talk` session, type `^c`.

`talk` sometimes does not work between different architectures. Users can deny permission to be talked to using the `mesg` command.

A.37 tar

```
tar key options file-list
```

creates an archive file or retrieves files from an archive file. The key determines whether to create or retrieve:

- **c** create an archive file
- **x** extract files from archive
- **t** list contents of archive

The most important options are:

- **v** verbose; tells what files are processed
- **f** indicates that the next argument is a filename to read from or write to; if instead of a filename a - sign is given, standard input or standard output is used
- **h** in creating, follow symbolic links and include the pointed–to files

To create an archive file from your C subdirectory in your homedirectory, give the following command in your homedirectory:

```
tar cvf C.tar C
```

This results in an archive file `C.tar`. To retrieve its content, type

```
tar xvf C.tar
```

A.38 tee

```
tee file-list
```

copies standard input to standard output and to the files given in file–list. To append to files, use the -a option.

```
a.out | tee outfile
```

displays the results of the executable `a.out` on the screen, simultaneously writing them to `outfile`.

A.39 uptime

displays the status of your machine: how long it's running, how many users are logged on and the load.

A.40 wc

```
wc file-list
```

counts lines, words and characters of the files given in file–list and displays the results. To have only one or two of lines, word and characters counted, give the options `l`, `w` or `c` or a combination of them.

A.41 which

```
which filename
```

displays the pathname of `filename`. `which` looks only in the directories specified in the `PATH` variable.

A.42 who

displays the users currently logged on the machine where you give the command.

B GNU Emacs Reference Card

In this reference card, `C-k` means "press the Control key and the `k` key simultaneously. `M-k` means "press the Meta key and the `k` key simultaneously. The Meta key can be marked `Meta` or ◇. If these keys are not present on your keyboard, the `ESC` key will work, too. In this case, press `ESC` followed by the appropriate key.

B.1 Starting and Leaving Emacs

To enter Emacs, just type its name:	`emacs`
To read in a file to edit, see Files, below.	
suspend Emacs (the usual way of leaving it)	`C-z`
exit Emacs permanently	`C-x C-c`

B.2 Files

read a file into Emacs	`C-x C-f`
save a file back to disk	`C-x C-s`
insert contents of another file into this buffer	`C-x i`
replace this file with the file you really want	`C-x C-v`
write buffer to a specified file	`C-x C-w`
run Dired, the directory editor	`C-x d`

B.3 Getting Help

The Help system is simple. Type `C-h` and follow the directions. If you are a first-time user, type `C-h t` for a **tutorial**. (This card assumes you know the tutorial.)

get rid of Help window	`C-x 1`
scroll Help window	`ESC C-v`
apropos: show commands matching a string	`C-h a`
show the function a key runs	`C-h c`
describe a function	`C-h f`
get mode-specific information	`C-h m`

B.4 Error Recovery

abort partially typed or executing command	`C-g`
recover a file lost by a system crash	`M-x recover-file`
undo an unwanted change	`C-x u` or `C-_`
restore a buffer to its original contents	`M-x revert-buffer`
redraw garbaged screen	`C-l`

B.5 Incremental Search

search forward	`C-s`
search backward	`C-r`
regular expression search	`C-M-s`

Use `C-s` or `C-r` again to repeat the search in either direction.

exit incremental search	`ESC`
undo effect of last character	`DEL`
abort current search	`C-g`

If Emacs is still searching, `C-g` will cancel the part of the search not done, otherwise it aborts the entire search.

B.6 Motion

Cursor motion:

entity to move over	**backward**	**forward**
character	`C-b`	`C-f`
word	`M-b`	`M-f`
line	`C-p`	`C-n`
go to line beginning (or end)	`C-a`	`C-e`
sentence	`M-a`	`M-e`
paragraph	`M-[`	`M-]`
page	`C-x [`	`C-x ]`
sexp	`C-M-b`	`C-M-f`
function	`C-M-a`	`C-M-e`
go to buffer beginning (or end)	`M-<`	`M->`

Screen motion:

scroll to next screen	`C-v`
scroll to previous screen	`M-v`
scroll left	`C-x <`
scroll right	`C-x >`

B.7 Killing and Deleting

entity to kill	**backward**	**forward**
character (delete, not kill)	`DEL`	`C-d`
word	`M-DEL`	`M-d`
line (to end of)	`M-0 C-k`	`C-k`
sentence	`C-x DEL`	`M-k`
sexp	`M-- C-M-k`	`C-M-k`

kill **region**	`C-w`
kill to next occurrence of *char*	`M-z` *char*
yank back last thing killed	`C-y`
replace last yank with previous kill	`M-y`

B.8 Marking

set mark here	`C-@` or `C-SPC`
exchange point and mark	`C-x C-x`
set mark *arg* **words** away	`M-@`
mark **paragraph**	`M-h`
mark **page**	`C-x C-p`
mark **sexp**	`C-M-@`
mark **function**	`C-M-h`
mark entire **buffer**	`C-x h`

B.9 Query Replace

interactively replace a text string	`M-%`
using regular expressions	`M-x query-replace-regexp`

Valid responses in query-replace mode are

replace this one, go on to next	`SPC`
replace this one, don't move	`,`
skip to next without replacing	`DEL`
replace all remaining matches	`!`
back up to the previous match	`^`
exit query-replace	`ESC`
enter recursive edit (`C-M-c` to exit)	`C-r`

B.10 Multiple Windows

delete all other windows	`C-x 1`
delete this window	`C-x 0`
split window in 2 vertically	`C-x 2`
split window in 2 horizontally	`C-x 5`
scroll other window	`C-M-v`
switch cursor to another window	`C-x o`
grow window taller	`C-x ^`
shrink window narrower	`C-x {`
grow window wider	`C-x }`
shrink window shorter	`M-x shrink-window`
select a buffer in other window	`C-x 4 b`
find file in other window	`C-x 4 f`
compose mail in other window	`C-x 4 m`
run Dired in other window	`C-x 4 d`
find tag in other window	`C-x 4 .`

B.11 Formatting

indent current **line** (mode-dependent)	`TAB`
indent **region** (mode-dependent)	`C-M-\`
indent **sexp** (mode-dependent)	`C-M-q`
indent region rigidly *arg* columns	`C-x TAB`
insert newline after point	`C-o`
move rest of line vertically down	`C-M-o`
delete blank lines around point	`C-x C-o`
delete all whitespace around point	`M-\`
put exactly one space at point	`M-SPC`
fill **paragraph**	`M-q`
fill **region**	`M-g`
set fill column	`C-x f`
set prefix each line starts with	`C-x .`

B.12 Case Change

uppercase word	`M-u`
lowercase word	`M-l`
capitalize word	`M-c`
uppercase region	`C-x C-u`
lowercase region	`C-x C-l`
capitalize region	`M-x capitalize-region`

B.13 The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	`TAB`
complete up to one word	`SPC`
complete and execute	`RET`
show possible completions	`?`
abort command	`C-g`

Type `C-x ESC` to edit and repeat the last command that used the minibuffer. The following keys are then defined.

previous minibuffer command	`M-p`
next minibuffer command	`M-n`

B.14 Buffers

select another buffer	`C-x b`
list all buffers	`C-x C-b`
kill a buffer	`C-x k`

B.15 Transposing

transpose **characters**	`C-t`
transpose **words**	`M-t`
transpose **lines**	`C-x C-t`
transpose **sexps**	`C-M-t`

B.16 Spelling Check

check spelling of current word	`M-$`
check spelling of all words in region	`M-x spell-region`
check spelling of entire buffer	`M-x spell-buffer`

B.17 Tags

find tag	`M-.`
find next occurrence of tag	`C-u M-.`
specify a new tags file	`M-x visit-tags-table`
regexp search on all files in tags table	`M-x tags-search`
query replace on all the files	`M-x tags-query-replace`
continue last tags search or query-replace	`M-,`

B.18 Shells

execute a shell command	`M-!`
run a shell command on the region	`M-|`
filter region through a shell command	`C-u M-|`
start a shell in window `*shell*`	`M-x shell`

B.19 Rmail

scroll forward	`SPC`
scroll reverse	`DEL`
beginning of message	`.` (dot)
next non-deleted message	`n`
previous non-deleted message	`p`
next message	`M-n`
previous message	`M-p`
delete message	`d`
delete message and back up	`C-d`
undelete message	`u`
reply to message	`r`
forward message to someone	`f`
send mail	`m`
get newly arrived mail	`g`
quit Rmail	`q`
output message to another Rmail file	`o`
output message in UNIX-mail style	`C-o`
show summary of headers	`h`

B.20 Regular Expressions

The following have special meaning inside a regular expression.

any single character	`.` (dot)
zero or more repeats	`*`
one or more repeats	`+`
zero or one repeat	`?`
any character in set	`[ ... ]`
any character not in set	`[^ ... ]`
beginning of line	`^`
end of line	`$`
quote a special character c	`\`c
alternative ("or")	\—
grouping	`\( ... \)`
nth group	`\`n
beginning of buffer	``\` ``
end of buffer	`\'`
word break	`\b`
not beginning or end of word	`\B`
beginning of word	`\<`
end of word	`\>`
any word-syntax character	`\w`
any non-word-syntax character	`\W`
character with syntax c	`\s`c
character with syntax not c	`\S`c

B.21 Registers

copy region to register	`C-x x`
insert register contents	`C-x g`
save point in register	`C-x /`
move point to saved location	`C-x j`

B.22 Info

enter the Info documentation reader	`C-h i`

Moving within a node:

scroll forward	`SPC`
scroll reverse	`DEL`
beginning of node	`.` (dot)

Moving between nodes:

next node	`n`
previous node	`p`
move **up**	`u`
select menu item by name	`m`
select *n*th menu item by number (1–5)	*n*
follow cross reference (return with `l`)	`f`
return to last node you saw	`l`
return to directory node	`d`
go to any node by name	`g`

Other:

run Info **tutorial**	`h`
list Info commands	`?`
quit Info	`q`
search nodes for regexp	`s`

B.23 Keyboard Macros

start defining a keyboard macro	`C-x (`
end keyboard macro definition	`C-x )`
execute last-defined keyboard macro	`C-x e`
append to last keyboard macro	`C-u C-x (`
name last keyboard macro	`M-x name-last-kbd-macro`
insert lisp definition in buffer	`M-x insert-kbd-macro`

B.24 Commands Dealing with Emacs Lisp

eval **sexp** before point	`C-x C-e`
eval current **defun**	`C-M-x`
eval **region**	`M-x eval-region`
eval entire **buffer**	`M-x eval-current-buffer`
read and eval minibuffer	`M-ESC`
re-execute last minibuffer command	`C-x ESC`
read and eval Emacs Lisp file	`M-x load-file`
load from standard system directory	`M-x load-library`

B.25 Simple Customization

Here are some examples of binding global keys in Emacs Lisp. Note that you cannot say `"\M-#"`; you must say `"\e#"`.

```
(global-set-key "\C-cg" 'goto-line)
(global-set-key "\e\C-r" 'isearch-backward-regexp)
(global-set-key "\e#" 'query-replace-regexp)
```

An example of setting a variable in Emacs Lisp:

```
(setq backup-by-copying-when-linked t)
```

B.26 Writing Commands

```
(defun <command-name> (<args>)
  "<documentation>"
  (interactive "<template>")
  <body>)
```

An example:

```
(defun this-line-to-top-of-screen (line)
  "Reposition line point is on to the top of
the screen.  With ARG, put point on line ARG.
Negative counts from bottom."
  (interactive "P")
  (recenter (if (null line)
                0
              (prefix-numeric-value line))))
```

The argument to `interactive` is a string specifying how to get the arguments when the function is called interactively. Type `C-h f interactive` for more information.

C Vi Reference Card

This section has been inspired by Sobell's "Practical Guide to the UNIX System"

C.1 Units of Measure

Commands in command mode often specify units of measure where the command shall take effect. These are

character	each character, including `SPACES`, `TABS`, punctuation and Control-characters
word	string of characters bounded on both sides by one or more punctuation marks, `SPACE`s, `TAB`s, digits or `NEWLINE`s. A group of punctuation marks is a word.
blank delimited word	the same as a word, but includes adjacent punctuation. Blank delimited words are therefore separated by so-called whitespace, i.e. `SPACE`, `TAB` or `NEWLINE`.
line	a string of characters bounded by a newline. This logical line is not necessarily the same as a physical line on the screen. If no `RETURN` is entered to end a line, `vi` continues this logical line. Some setups of `vi` automatically separate logical lines into physical lines.
sentence	a string of characters bounded by a period, an exclamation point or a question mark followed by two (!) `SPACES` or a newline.
paragraph	a string of characters followed by at least one blank line.

If you precede a unit of measure with a number, this number is taken as a repeat factor for the unit of measure.

C.2 Starting and Leaving vi

`vi`	starting `vi` without specifying an input file
`vi filename`	edit file `filename`
`vi -r filename`	recover `filename` after a system crash
`:wq`	save changes and quit `vi`
`ZZ`	save changes and quit `vi`
`:q!`	discard changes and quit `vi`

C.3 Moving

Remember that you have to be in Command Mode to use the following commands for moving the cursor. They also specify the Units of Measure you can use with other commands. Each command may be preceded by a number n which causes repetition of the command n times.

Note that the keys `h, j, k, l` which are right under your fingers while you're typing have the same meaning as cursor keys.

`h`	character to the left
`j`	down one line
`SPACE, l`	character to the right
`k`	up one line
`w`	word to the right
`W`	blank delimited word to the right
`b`	word to the left
`B`	blank delimited word to the left
`e`	end of word to the right
`E`	end of blank delimited wird to the right
`0`	beginning of line (cannot be used with a repeat factor)
`RETURN`	beginning of next line
`-`	beginning of previous line
`$`	end of line
`(`	beginning of sentence
`)`	end of sentence
`{`	beginning of paragraph
`}`	end of paragraph
n`G`	to line *n* (without *n* to the last line)
`H`	to top of screen
`M`	to middle of screen
`L`	to bottom of screen

The next commands move the cursor by bigger portions of the file. These are no Units of Measure.

`^d`	down half a screen
`^u`	up half a screen
`^f`	forward one screen
`^b`	back one screen

C.4 Inserting Text

With the following commands you stay in Insert Mode until you press `ESC` to return to Command Mode.

The following commands insert text

`i`	before cursor
`I`	before first nonblank character on line
`a`	at current cursor position
`A`	at end of line
`o`	open line below current line
`O`	open line above current line

C.5 Deleting

In this section and the following ones, *M* is a Unit of Measure as given in section C.3 that can be preceded by a repeat factor *n*. Note that `d` followed by `RETURN` deletes two lines, the current line and the following one. To delete the current line only, use `dd`.

n`x`	delete *n* characters, starting with current one
n`X`	delete *n* characters before current one, starting with character preceding the current one
`d`*M*	delete text specified by *M*
n`dd`	delete number of lines specified by *n*
`D`	delete to end of line

Examples:

`d0`	delete to beginning of line
`dW`	delete to end of blank delimited word
`5dd`	delete 5 lines starting with current line
`dG`	delete through end of file
`d1G`	delete through beginning of file

C.6 Changing

Giving one of the first four commands in the following list, you will see a `$` sign at the end of the Unit of Measure specified. Everything from the cursor to the `$` sign will be replaced by what you enter until you press `ESC` to return to Command Mode. If you reach the `$` sign while entering text, `vi` will continue to insert the characters you type.

n`s`	substitute *n* characters
`c`*M*	change text specified by *M*
n`cc`	change *n* lines
`C`	change to end of line
n`r`*c*	replace *n* characters by the single character *c* (returns to Command Mode)
`R`	replace (overwrite) text until `ESC` is pressed

C.7 Searching for a String

In the following, *regex* is a regular expression (see appendix D) that can be a simple string of characters.

/*regex* `RETURN`	search forward for *regex*
?*regex* `RETURN`	search backward for *regex*
`n`	repeat original search in same direction
`N`	repeat original search in opposite direction
`/RETURN`	repeat original search forward
`?RETURN`	repeat original search backward

C.8 String Substitution

A substitution command has the following syntax:

```
:[address]s/search-string/replace-string/g
```

`address`	consists of two line numbers separated by a comma. The line numbers can be replaced by a `.` (dot), representing the current line, a `$` representing the last line or a mark sign (see section C.11).
`search-string`	is a regular expression that can be a simple character string
`replace string`	is a regular expression that can be a simple character string
`g`	indicates global replacement; if you omit the `g`, only one replacement per line will be performed.

C.9 Yanking Text

Yanking in `vi` means "remember, but do not delete". You can put yanked text somewhere else in the file. To store it for later purposes, you have additional buffers available, named from a to z. Just precede the yanking command by `"x` where `x` is the buffer name.

`y`*M*	yank text specified by *M*
n`yy`	yank *n* number of lines
`Y`	yank to end of line

C.10 Putting Text

After you deleted or yanked text, you can put it somewhere in the file. If you did not specify a buffer for intermediate storage, you may only perform cursor motions between the yank/delete command and the put command. To put something stored in buffer `x`, precede the put command by `"x`.

`P`	put text before cursor
`p`	put text after cursor

C.11 Marking

`m`*x*	set mark *x* where *x* is a letter from a to z
'*x*	move cursor to beginning of line containing mark *x*
‘*x*	move cursor to position of mark *x*
`''`	move cursor to previous location

C.12 Shell Commands

`:sh`	create a (sub)shell; return to editor with `^d` or `exit`
`:!command`	create a shell and execute `command`
`!!command`	create shell, execute `command` and place the standard output of `command` in the file you are editing, replacing the current line
`:r!command`	insert the output of `command` at current position of cursor

C.13 Miscellaneous Commands

`u`	undo last change; works only once!
`J`	join the current and the following line
`.`	repeat the most recent command that made a change
`:w file`	write changes to `file` (current file if no `file` has been given as argument)
`:e file`	edit `file`; use `:w` first to store the changes you made to the current file before reading in a new one.
`:f`	display name and status of current file
`:r file`	insert file at current position of cursor
`^g`	display filename, current line number, total number of lines and the percentage of the file preceding the current line
`^v`	insert next character literally. This is used in Input Mode to insert Control-characters etc. literally
`~`	change uppercase to lowercase and vice versa

D Simple Regular Expressions

Regular expressions are used in several UNIX utilities and editors for searching and replacing sets of strings. This chapter is restricted to the simple regular expressions as used by `vi`. For the full regular expressions, you should have a look at the manpage for `egrep` or section B.20. The usage of regular expressions in `emacs` is somewhat different then described here. Look at the appropriate parts of the `emacs` manual.

A regular expression can be a simple string or a combination of characters which matches a set of strings. This is achieved by giving some characters a special meaning. If you want to use these special characters in a regular expression literally, you have to quote them by preceding them with a backslash. \\ represents a literal backslash.

You have seen this type of referring to a set of strings already in the shell's filename expansion. There, `*`, `?`, `[` and `]` had special meaning. Regular expressions are more complicated and powerful.

In most cases, the regular expression has to be bracketed between two equal characters. They can be anything as long as they are the same. These delimiters are simply defined by the first character that appears which is taken to be the delimiter. The delimiter is special to this regular expression and has to be quoted if appearing inside the regular expression. Often, a `/` is used as delimiter, and I will use this in the following.

The simplest regular expression is a simple string. It matches only itself. `/abc/` would match anything like `abc`, `aabcc` etc.

Special characters are:

- **. (period)** matches any single character (like `?` in the shell's filename expansion)
- `[ ]` equivalent to the shell's filename expansion. This matches a single character being in the list (like `[abc]`) or range (like `[a-c]`) given in the brackets. If the left bracket is followed by a `^`, every character not listed in the brackets will match. The special charcters `\, *, $` lose their special meaning (see below). A `^` is special only as the first character following a left bracket. The `-` sign is used to indicate a range and therefore special, except as first character after `[` or `[^`. If you want to include a `]` in the list, you have to quote it as `\]`.
- `*` An asterisk after a regular expression which represents a single character means that this regular expression will be matched if occuring zero or more times. `ab*c` would match `ac`, `abc`, `abbc` etc.

An asterisk following a period matches any string of characters. This is equivalent to the single asterisk in the shell's filename expansion.

- **^ (caret)** A regular expression beginning with a caret matches only a string at the beginning of a line.
- **$** A regular expression ending with a **$** matches only a string at the end of a line.
- **\(and \)** Quoted parentheses can be used to group parts of an expression. These groups can be referred to later by quoted digits.

The following two are special to searching and replacing in `vi` and `sed`:

- An ampersand in the replacement string is replaced by the string the search string matched.
- In a search string a quoted digit (`\n`) represents the string of the `n`th expression that has been grouped in the search string by quoted parenteses.
- In a replacement string a quoted digit refers to the `n`th grouped regular expression that has been matched in the search string.

To avoid ambiguities there are the following rules:

- A regular expression matches always the longest possible string.
- An empty regular expression represents the last regular expression used. An empty regular expression is just one with two equal characters, namely the two delimiters.

E Literature

Here are some books you might find useful.
A nice introduction to UNIX is

- Mark G. Sobell, A Practical Guide to the UNIX System, Benjamin/Cummings Publishing Company, 1989.

If you want to learn C, the following books are recommended

- Mitchell Waite and Stephen Prata, The Waite Group's New C Primer Plus, Howard W. Sams & Company, 1990
- Brian W. Kerninghan and Dennis M. Ritchie, The C Programming Language, 2nd edition, Prentice-Hall

The definite introduction to the X window system is O'Reilly's X Window Series. The volume most suited for a beginner is

- X Window System User's Guide, The X Window System Series, O'Reilly & Associates, 1990

If you want to gain much (!) deeper knowledge about UNIX and networking, try

- S.J. Leffler et al., The Design and Implementation of the 4.3 BSDK UNIX Operating System, Addison–Wesley, 1990
- W. Richard Stevens, UNIX Network Programming, Prentice–Hall, 1990