

Unix/Linux has a reputation in certain circles of being a hostile world where even the simplest commands have been deliberately hidden from prying eyes by choosing cryptic names, and further protected by writing the manual in such a clever way that you must know the answer to ask a question. I must admit that unix command names *are* concise (and that vowels seem to have been especially expensive), but most of the difficulty is due to unfamiliarity: is it that much clearer to say “`set def [.rh1]`” than “`cd rh1`”, and is “`mms all`” that much more intuitive than “`make all`”?

This document introduces you to unix’s commoner commands and some of its glories (such as redirection and piping), and points you in the direction of a few of the commands that make unix such a wonderfully rich and user friendly environment[†].

If you have problems you can’t solve by perusing this manual you should ask for help. Possible saviours are fellow students, grad students, and our systems folks Leigh Koven and Steve Huston. You should send email requesting help to `help@astro.princeton.edu`.

Logging in

Enter your username and password (if you were given one), and you’ll get your first unix prompt. Any word that you type will be taken to be a command – you don’t need hieroglyphs such as `@` to indicate that you want to execute a command file.

passwd Please change your password to something suitably obscure! Type `passwd -y` (followed by a carriage return; all commands are executed with a carriage return, which I shall not mention again). You’ll be prompted for your old password, then your new one, then your new one again to ensure that you typed it correctly. We are not especially worried about security here, but it is still not a good idea to choose your cat’s/wife’s/husband’s/lover’s name, nor any word in the dictionary.

man If you want to know more about a given command use the on-line manual, *e.g.* `man passwd`. The man pages (as they are known) tend to be concise, but complete. If you don’t know the name of the command use `man -k string` (‘keyword’) which searches the one-line titles of all the manual entries looking for your string. The man pages cover the programming libraries as well as commands, for example `man -k ieee` asks what the system knows about IEEE arithmetic. The numbers that appear in parentheses after command names give the section that the command appears in, section 1 consists of user commands, sections 2 and 3 are programming libraries, sections 4 and 5 specify file formats, section 6 is games, section 7 is of little interest, and section 8 describes commands primarily of interest to system people. Please remember the `-k` flag to `man` — if you don’t know the name of a command you can usually at least guess a keyword that appears in its description.

The environment variable `MANPATH` can be used to specify where `man` should look for manual pages; it is set to a suitable default value in the standard startup files.

[†] I might, however, be prepared to agree that it isn’t especially beginner friendly.

There are printed copies of the on-line manuals, as well as more extensive and even tutorial discussions of some commands, in the undergraduate room (Room 29). There are also copies of some books about unix, languages, window systems and so forth. Please remember to sign for any that you borrow.

logout To end a session type `logout`. Alternatively, you can logout with `^D`[‡] (unless you have chosen to disable this feature). `^D` is the ASCII End-Of-Transmission character and unix consistently uses it to mean, “I am finished”. Do **not** try to use `^Z` for this purpose, oh User of VMS, as it won’t work (even if you think that it did, it didn’t. If you want to know what it *did* do, see the section ‘More about Shells’, specifically where the special characters `&%;` are discussed.).

Shells

When you log in to a unix system you aren’t talking to ‘unix’, but to a command interpreter called a *shell*. I’ll postpone a discussion of their properties and merely note that they treat certain characters specially, specifically `/ < > | & % ; " ' \ $ () { } [] ? * ~ !` and `^`, and you should try not to use them in filenames. For now it’s enough to know that `/` separates directory names from filenames (*e.g.* `dir1/dir2/file`) and `*` is a wildcard that matches any string, so `*.c` expands to a list of all files whose names end in “.c”. If other characters in this list (such as `|`) appear in the following descriptions be patient, or skip ahead to the other section on shells. The names of the shells are `bash`, `csh`, `tsh`, and `sh` so if I refer to `(t)csh` in the command descriptions I mean ‘`tsh` or `csh`’. The character `-` is not special, but it is conventionally used to introduce options to commands.

Files

ls You can list your files with `ls`. As is true of many unix utilities, `ls` has many options, of which some of the most useful are `ls -F` to indicate the type of file (executable, directory, etc.), `ls -l` to give a long listing, and `ls -A` to include files starting with a `.` which are usually not shown. For example, `ls -F a*` will list all files starting with an ‘a’, and `ls -F [A-Z]* | grep / | more` lists all the directories starting with capital letters (the `grep` command is discussed later, or look it up in the manual).

You have a choice of editors, with most people choosing either `vi` or `emacs`. They all use the value of the environment variable `TERM` to tell them what sort of terminal you are on, and if it is wrong the editors won’t work correctly.

vi The standard unix editor is `vi` which is a 2-mode editor (at any moment you are either able to enter text or to move around the screen, but not both). It’s a very powerful and fast editor once learnt, but some people claim that it is impossible to learn. It is available on all unix systems.

emacs Emacs is in fact a family of editors, we mostly use one called `Gnuemacs` which is widely held to be the most powerful editor in the world. Emacs is a mode-less editor unlike `vi`, everytime that you hit a printing character it appears on your screen. The price

[‡] *i.e.* ‘control-D’, hit the ‘control’ key and ‘d’ simultaneously

you pay for power is complexity, and some neophytes find the sheer size of the editor's command set intimidating. Emacs can be run on almost any computer (even VMS vaxes), but it is not usually provided with the operating system, so you could find yourself on a machine without emacs (although you could bring a source tape).

Now that you have some files you need the various commands to look at them, move them around, and delete them again.

cat Once you have a file you can look at it with `cat` which types it on your terminal. The name comes from its use in concatenating files, *e.g.* `cat file1 file2 ... > newfile`.

more If you don't want your file to fly off the top of the screen as you try to read it `more file` will present it a page at a time. A common use of `more` is at the end of a pipeline, *e.g.* `ls -l -C | more` (see 'More about Shells' if you are confused). A similar programme with slightly different capabilities is `less`.

rm To remove a file or files use the `rm` command. One of its most popular arguments is `-i` ('interactive'), in which case it will query you before removing a file. Some users even like to alias it, `alias rm '/bin/rm -i'` ((t)osh) or `rm () { command rm -i $* }` (bash). Because all wildcards are expanded by the shell before the command ever sees them a command such as `rm *.f` will indeed remove all files ending '.f'.

There is no `unrm` command — once a file is deleted it is gone. If you do lose some vital file don't be downhearted, we backup your home directory to tape every night so you can ask Jay, Jeremy, or me to restore your files.

mv To move a file from one place to another (which is usually just a simple renaming) use `mv name newname`. You can just as well rename a directory with `mv dir newdir` or move a number of files to a new directory with `mv file file ... dir`. A popular option is `-i` which queries you before overwriting a pre-existing file.

cp To make a copy of a file, or a number of files, use the `cp` command. To copy a complete directory tree you must specify the `-r` ('recursive') flag. The `-i` option is another perennial favourite.

Directories

When you log on you are in your home directory (which can be referred to as `~` or `$HOME`). Unix supports a hierarchical directory structure which provides a way to organise your files in some way other than putting them all your home directory.

pwd To see where you are use `pwd` ('Print Working Directory'). Some people like to put at least part of their current directory name into their prompt; techniques to do this are left as an exercise for the reader.

cd To change directories use the command `cd name`, where `name` is usually taken to be a subdirectory of the current directory, which is known as `.` if you want to refer to it explicitly. The directory above `.` is called `..` (*i.e.* `.` is a subdirectory of `..`). For

example, `cd ../innuendo` will move to a directory called `innuendo` at the same level as your current directory. If `name` starts with a `/` it is taken to be an absolute path name, *e.g.* `cd /usr/include` and if you omit `name` you'll end up in your home directory.

mkdir To make a new directory use the command `mkdir name`. If the new name doesn't start with a `/` it is taken relative to the current directory.

rmdir To delete a directory say `rmdir name`. If it isn't empty `rmdir` will refuse to delete it. A convenient way to destroy complete directory trees is to use the `-r` ('recursive') flag to `rm`, for example `rm -rf ~rhl/*`. You can use the `man` command to discover the significance of the `f` flag.

Languages

Unix programmers have traditionally used C as their main language, but we also have a fortran-77 compiler. We do not currently have support for either pascal or modula-[23]. Unix compilers are also used as linkers, so there is usually no need to explicitly call the loader (whose name is `ld` if you ever need to look up its man page). The commands to compile the fortran programme found in `file1.f` and `file2.f` are:

```
f77 -c file1.f
f77 -c file2.f
f77 -o outfile file1.o file2.o
```

where the `-c` option means 'compile, don't link' and `-o` specifies the name of the executable (which defaults to 'a.out') (to compile a C programme replace the ".f"s with ".c"s, and change `f77` to `cc`). To run your programme simply type its name: `outfile`. If you want to run the optimiser specify `-O`, if you want to keep debugging information specify `-g` (some compilers do not allow both).

cc The system C compiler is called `cc` and expects source filenames to end in ".c". It is not ANSI compliant, but does produce good code (at least on the Sun-4's). If you use any mathematical functions you must request the math libraries with `-lm` at the end of the last line of the example (*i.e.* the line that actually links your programme).

gcc We also have the GNU C compiler `gcc` which is a re-targetable, public domain, highly optimising, ANSI compliant, compiler. It also expects filenames to end in ".c". It may not produce quite such good code for floating-point intensive code as `cc` on a Sun-4 (but I use it anyway). There is also `g++` which is the GNU C++ compiler.

f77 The standard unix fortran compiler is called `f77`, understands Fortran-77, and expects filenames to end ".f" or ".for". It will accept almost all VMS fortran extensions, although using them in new code is not a good idea.

fc The convex fortran compiler is called `fc`, expects filenames to end ".f", and has a number of levels of optimisation, of which `-O3` is the most aggressive.

make It is of course easy to write shell scripts (or even aliases or shell functions) that compile and link a source file, but there is a much better way. The programme `make` takes a description of your source files (called a *makefile*) and decides what files need

re-compiling and re-linking after you make a change. For example, imagine that I have just changed a file that some of my source files `include`, and I now want to recompile only the files that are affected. I can do this by typing “make”, which recompiles all the source files that include that file, rebuilds any libraries that include the new object files, and relinks any executables that include those libraries. When I have 30,000 lines of code spread over 5 directories this is convenient, but it is also worth doing for small projects, if only because it provides documentation of which libraries are needed.

As an exception to my usual advice to read the manual (an expression usually abbreviated RTFM), I recommend that you learn to use `make` by borrowing someone else’s Makefile, and *then* reading the manual.

Printing

We have three printers in Peyton hall, called `imagen`, `ps`, and `fred`. Printers are specified by preceding their name by `-P` when you use any of the following ‘lp*’ commands (*e.g.* `lpr -Pfred filename`). Alternatively you can set the environment variable `PRINTER` to one of these to make it your default printer in which case you don’t need to use `-P`. For those of you who have resisted the temptation to skip ahead, this is achieved with `setenv PRINTER imagen ((t)csh)` or `PRINTER=imagen ; export PRINTER ((ba)sh)`.

lpr The simplest way to print a file is to say `lpr file`. This is almost equivalent to `cat file | lpr`, demonstrating that you can use `lpr` in pipelines. To print a dvi file (typically produced by \TeX) use `lpr -d file.dvi`.

lpq To list the printer queue use `lpq`.

lprm To remove a job from the print queue use `lprm number` where `number` is given by the `lpq` command.

Typesetting

You have a choice of two typesetting programmes, `nroff` and \TeX . The unix standard is `nroff` (and variants such as `vtroff`), while the astronomical standard is \TeX . I recommend that you learn \TeX .

nroff There are a number of varieties of `nroff` which print to different devices: `nroff` (terminals), `ptroff` (postscript printer), `itroff` (`imagen`), `ditroff` (device independent), and many more. You also have your choice of macro packages, the commonest being `-me`, `-ms` and `-man`. The task of making tables and typesetting mathematics has been split off into two separate programmes connected by pipelines, so the canonical incantation is something like `tbl file | eqn | ptroff -ms`.

\TeX To run \TeX on a file say `tex file.tex` or `tex file`. This produces a file `file.dvi`, which you can print using `lpr -d file.dvi`, and a file `file.log`, which you can safely delete.

Processes and Users

There are a number of commands (to be discussed in the section ‘More about Shells’, specifically under `&%;`) to suspend and restart jobs, and to run them in the ‘background’ while you are busy doing other things. Sometimes these are not what you want, in which case you may need to know about:

ps The command `ps` lists all of your current processes. There are a number of common flags: `-a` lists everyone’s processes, `-g` lists them even if they are not attached to a terminal, `-l` gives a long listing, and `-u` lists them by user. A common command is thus `ps -au | more`.

kill To kill a process you need its ‘pid’ (Process ID) number, which you can get from `ps`. Then you either say `kill pid` or `kill -n pid` where `n` is a number which specifies the mode of execution. A popular choice is `kill -9` for which no clemency is possible; a process may choose to ignore some `kill` commands, but not the dreaded ‘-9’. A common use for `kill` is when you have hung your terminal; log in somewhere else, use `ps g` to find the offending programme’s pid (maybe your shell itself), and destroy it with `kill -9 pid`.

w Type `w` to see who else is logged on. As an alternative, you might try `who` or `who am i`.

finger Who is that sinister-sounding user `jpo`? Type `finger jpo` to find out.

Mail

The mail system is “pine”. Type `pine`. This allows you to read your mail, reply (`r`), forward (`f`) and compose (`c`). To send a mail message, hit control `x`. There is help available within `pine`.

The Web

Click on your browser (or type `mozilla &`). The `&` puts the task in background. To access a site, say for example `http://www.astro.princeton.edu` which gets the astro home page.

To access the literature

click on library click on Astrophysical Data System to get you into the published literature click on arXiv.org to get the preprint server.

Printing

Plain ascii text: `lpr -Pfred filename.txt` (fred is the printer in Rm 29)

Postscript: `lpr -Pfred filename.ps`

HOWEVER: if the file is *not* plain text or postscript DO NOT use `lpr`; you’ll get piles of paper. To check, say

more filename.txt if the file is plain ascii

gv filename.ps if the file is postscript

If you can read them OK, they're safe to print.

To print a pdf file, say `acroread filename.pdf` and click on print

Searching

There are two common sorts of searches; searching for a string in a known set of files, and searching for an unknown file. Sometimes, of course, you need to do both at once.

grep The command to find a string in a file (or files) is `grep` ('Global Regular Expression Print', a name surviving from an ancient editor). The simplest version is `grep string file1 file2 ...`, which looks for the string in each of the files specified; if none are specified it reads its standard input so it can be used in a pipeline. If you want to ignore case, or look for `string` as a word, or only at the start of a line, or whatever you desire, read the manual.

find If you want to find a file called `file` anywhere in the directory tree starting at `dir` you should say `find dir -name file -print`, or to find all the C source files, use `find dir -name *.c -print`. If this seems complicated, consider that I can delete all files called `core` more than 3 days old and larger than 1Mby by saying:

```
find dir -name core \( -atime +3 -o -size +2000 \) -exec rm \{\} \;
```

If I want to combine the two types of search I can use something like:

```
find dir -exec grep word /dev/null \{\} \;
```

(of course, if I just wanted to search a directory or two I'd use `grep word dir1/* dir2/*`).

More about Shells

There are currently four shells in use at Peyton Hall, `sh` (written by, and named after, Steve Bourne of AT&T), `csh` (the 'C-shell'), `tcsh` (an improved version of `csh`), and `bash` (The Bourne Again SHell). Although in your first exposure to a shell it's in its 'command interpreter' guise, in fact shells are also powerful programming languages. There is nothing special about a shell, by the way, it is just a programme like any other, so if you want to write your own shell go ahead. A consequence of this is that if you want to switch to another shell all you have to do is run it — type its name to run it as you would for any other command.

Mostly you don't have to worry about which shell to use. A few commands are different between (t)csh and (ba)sh (many more if you write command scripts, in which case (ba)sh is generally preferred by the cognoscenti). Bash, csh, and tcsh support a *history* mechanism for recalling and repeating commands; bash and tcsh allow you to use arrow keys to retrieve commands (like VMS's history, but a good deal better).

All unix systems have sh, and almost all have csh so you might want to use one of these, in which case I would advise using csh. I personally use bash, and Jay uses tcsh.

.login When you log in your shell executes a startup file called variously `.login` ((t)csh), `.profile` (sh), or `.bash_profile` (bash). These ‘dot’ files are not usually listed by `ls` or matched by `*`. Every time that a new shell is created (for example, when you run a shell script or open a new window) another startup file is run, called variously `.cshrc` (csh), `.tcshrc` (tcsh), or `.bashrc` (bash). Things that you only want done when you log in (such as checking for new messages) should go in your “.login” file, things that you want done for every shell (such as setting up aliases (next paragraph)) should go in your “.cshrc” file (in fact it is common to reduce clutter by putting all your aliases in yet another file and saying `source .my_aliases` at the bottom of your “.cshrc” file).

alias You can use the `alias` command to rename or slightly modify your favourite commands. To make `goodbye` log you out you’d say `alias goodbye 'logout'` in (t)csh, or `alias goodbye='logout'` if conversing with bash (the `'` are optional, but recommended). You can include arguments in (t)csh aliases in a rather obscure way (e.g. `alias pdvi 'lpr -d \!*.*.dvi'`). Bash aliases don’t take arguments, but you can use the very useful *shell functions* instead: `lpdvi () { lpr -d $*.dvi }`.

You should note that aliases and shell functions are usually *not* the right way to execute your favourite programmes. It’s better to put them in a directory that appears in your `PATH` environment variable (`echo $PATH`), after which you can execute them by simply typing their name. The directory `~/bin` is already in your path for this very purpose. (Variables are discussed below under `$`).

All shells treat a number of characters specially, specifically `/ < > | & % ; " ' \ $ () { } [] ? * ~ !` and `^, †`. You are already familiar with `/`, which is used to separate directory and file names, and the rest can be broken up into six groups:

`< > | &` Are used to *redirect* a command’s input and output, specifically the *standard input* and *standard output* (file descriptors 0 and 1 to C programmers, units 5 and 6 to fortran). Unix also recognises *error output* (file descriptor 2) as you will soon see. For example, if your programme called `monkeys` produces output that you want to keep, you could type `monkeys > Hamlet` to save it in a file called `Hamlet` (this is a different file from `hamlet` — unix is case-sensitive). If you already have a file called `Hamlet` it will be overwritten, but you can append your output to a file with `monkeys >> folio ‡`. If your programme also produces error messages then these messages *won’t* go into `Hamlet`, but will instead be printed on your terminal. If you want to redirect error messages as well use the command `monkeys >& Hamlet`. The (t)csh doesn’t allow you any easy way to redirect the errors to a different file, but (ba)sh allows `monkeys > Hamlet 2> Bacon`.

If your programme expects to read from the terminal you can use `<` to use a file instead. For example, you will remember that the command to send me mail is `mail rh1` (after which you type your message). If instead you put the message into a file called `H2S04` you can send it to me with `mail rh1 < H2S04`. The form `<<` also means something (take the input from the current file), but that’s a bit advanced to go into here.

† actually sh doesn’t care about `{ }!` or `^`.

‡ If you set the variable `noclobber` a file won’t be over-written; (t)csh users can use `>!` to force the destruction.

Sometimes you want to send the output of a programme, not to a file, but to another programme, say `performance`. This is easily achieved with `monkeys | performance`, where the character `|` is usually pronounced pipe, and the whole composite command is called a pipeline. This sort of thing can be chained together without limit, *e.g.* `monkeys < bananas | performance > profit` where I have also specified an input file.

`&%`; You can run a programme in the *background* by putting an `&` at the end of the line. You can issue other jobs while your programme runs (if it spits blank verse all over the screen that's because you forgot to redirect its output with `>`). If you start a job in the foreground you can stop it by typing `^Z`, in response to which it'll print a message including the word 'stopped' and return you to the shell. Type `jobs` to get a list of jobs, either running or stopped, and with a number like `[2]` attached. You can now say `bg %2` or `fg %2` to put job number 2 into the background or foreground; simply typing `%2` is equivalent to `fg %2` and you can use any unambiguous abbreviation of the command name in place of the number, for example `bg %monk`.

You are not restricted to single commands, you can put a `&` at the end of a pipe to run it in the background, or separate a number of commands that should run sequentially with semicolons, for example `sleep 1800 ; mail rh1 < H2SO4 &` to give you half an hour to leave the building.

`[]*` When it sees one of these characters as part of a word the shell attempts to replace the word by all filenames that match the word treated as a pattern. The character `*` stands for zero or more characters (excluding `/` but including `.`, with the exception that a `.` at the start of a word isn't matched), so `*.c` will expand to all filenames ending `".c"`. *Please note* that the expansion is done by the shell, so a command like `mv *.f *.c` won't rename files ending `".f"` with the same name ending in `".c"`[†].

A `?` stands for exactly one character (again excluding `/` and a leading `.`). `[abcd]` stands for 'one of a, b, c, or d'. If the first character is a `^` it means 'except', so `[^abcd]` matches anything except a, b, c, or d and you can specify a range with a `-`, so `[0-9]` matches any digit.

A `~` by itself stands for your home directory so `~/file` is a file in your home directory. A `~` followed by a username is their home directory, so `~rh1/file` is a file in mine.

`!^` The (t)csh and bash allow you to reuse previous commands with `!` and `^`, while tcsh and bash also allow you to recover and modify old commands with editor-like keys (including, but not limited to, the arrow keys on your keyboard).

Each command has a number associated with it (you can list commands with `history`), and to repeat command number 22 you would say `!22`. The number of commands remembered is set by your `history` variable, which you set like any other variable (see the discussion of `$`, or try either `set history = 80` for (t)csh or `history=80` for bash). To repeat the last command beginning with 'string' say `!string`, to repeat the last

[†] to do that you need a `for` ((ba)sh) or `foreach` ((t)csh) loop.

command say `!!`, to get the last word of the previous line say `!$`, and to learn more see your shell's manual.

`\''` With all of this multitude of special characters there has to be a way to turn them off; in fact there are three ways, each of which is less powerful than the previous. Preceding any character with a `\` turns off its special meaning, enclosing a string in 'single quotes' turns off the special meanings of all characters except `!`, and "double quotes" are like single ones, except that they don't stop `$`, `'`, and `\` from being treated specially.

`$(){ }` All the shells support variables, in (ba)sh you set a variable by saying `name=value` while (t)csh demands `set name = value`. If you want to include spaces in your values use quotes, for example `dimen="31 42"`. A variable is used by preceding its name with a `$`, for example try the command `echo dimen: $dimen`. Some variable names are interpreted by the shells, *e.g.* `history` and some are used to pass information to other processes such as editors. These last (*environment*) variables are traditionally written in capitals and must be specially identified as needing to be sent to other processes. In (ba)sh this is achieved with `export NAME`, while (t)csh uses a different command to set them in the first place: `setenv NAME VALUE` (*n.b.*: no equals sign). The two most important environment variables are `TERM` which tells various programmes what sort of terminal you are on, and `PATH` which lists the directories which the shell should search for commands to execute.

Any command enclosed in 'back-quotes' is executed and the 'command' is replaced with its output. A command or set of commands that is enclosed in parentheses is run in a sub-shell, and a string can be enclosed in {braces} for a number of good reasons that I shan't go into here.

A Few More Commands

Here's a quick mention of a few of the commands awaiting you, use the manual to find out more about them.

awk Perform operations on the columns of a file (or pipeline), *e.g.*
`ps 1 | awk 'NR > 1 {RSS += $9} END {printf("Size = %d kby\n",RSS)}'`

clear Clear the screen.

date What's the date and time? You don't want the year? Use
`date | colrm 20 80`

dbx/gdb When a programme crashes and produces a `core` file use one of these debuggers for the postmortem (don't use `adb`). You must have compiled with the `-g` flag for them to be much help.

dd Transfer files, usually to/from tape, optionally doing things like swapping bytes along the way.

diff Find the differences between two files. The `-c` option provides a bit of context.

echo Print a string on your terminal.

foreach/for Set a variable to each of a list of words and execute a set of commands; `for` is used by (ba)sh, (t)osh uses `foreach`. For example, in (ba)sh:

```
for f in *.c; do mv $f $f.sav; done
```

if Execute some commands if a command succeeds or a condition is true. (The syntax differs between (t)osh and (ba)sh).

hack Bring the machine to its knees.

mt Manipulate magnetic tapes (rewind and so forth). To unix tapes are just files that happen to have names like `/dev/nrst0` rather than `nobel.c`.

ox2 Look up a word in the second edition of the Oxford English Dictionary. Non-native speakers may prefer to use **webster**.

rsh With just a machine-name (*e.g.* `rsh grendel`) login to another machine; if followed by a command (*e.g.* `rsh grendel who`) run a command on a remote machine.

sed Run a set of editing commands on a file or pipe; for example

```
echo file.f | sed 's/\.f/.c/'
```

or (more interestingly, as it could be in a `for` loop)

```
mv file.f 'echo file.f | sed 's/\.f/.c/''
```

In fact this particular example can be done more easily with

```
mv file.f 'basename file.f .f'.c
```

spell Check the spelling of a file (or standard input). Spell makes fewer mistakes with the `-b` option. You might prefer the interactive spell-checker `ispell`.

talk Establish two-way, split-screen, communication with another user.

tar To create a backup of a directory tree use `tar`; although the `t` stands for 'tape' it works perfectly well with pipes or files instead, *e.g.*

```
tar -xf - dir | compress > dir.tar.Z
```

tea How long is it until tea? See also `teanote`.

while While a condition is true (or a programme succeeds) execute some commands.