Unix Tools

Ian Pratt, Michaelmas Term 2000

Original notes by A C Norman

Part IB

4 Lectures

1 Syllabus

Unix Tools (4 lectures, no examination questions)

1. Overview and basics. Expected impact of this course on future CST practical work and the evaluation criteria applied to it. The Unix philosophy and history. Streams, redirection, pipes. Ways to identify information (wildcards, find, grep and the use of regular expressions). Influence of Unix tools on the non-Unix world.

2. Tools to support development and testing. Makefiles & parameter substitution. Important shell facilities including job control, "nice" and history-substitution.

3. Data manipulation tools: sed, tr, diff. Why these may be useful.

4. Scripting with perl and hence more advanced linkage between tool components.

Books:

1. *Unix in a Nutshell*, Daniel Gilly, O'Reilly

2. *Learning Perl (edn 2)*, Schwartz and Christiansen, O'Reilly

2 Introduction

This course is called "Unix Tools", and this is because the various support utilities that it discusses originated with Unix and fit in with a philosophy that was made explicit from the early days of that operating system. It should however be noted that most of the particular programs or tools mentioned have been found sufficiently useful by generations of programmers that versions have been ported to other operating systems, notably Microsoft Windows, and so whatever platform you use now or expect to use in the future there is something here that may prove relevant to you. Even when running under Windows it seems proper to refer to things as "Unix" tools, both for historical reasons and because the style of interface that these tools provide contrasts quite strongly with that seen in (say) the Microsoft Visual Studio[1].

[1] The environment within which one influential vendor's set of native Windows development tools reside.

This course is short and it is also unusual in that no questions on it will appear on the examination papers at the end of the year. These two facts may lead to the impression that the department considers the material covered unimportant or optional. Any such impression is ill-founded. It is anticipated that techniques mentioned during this course will be of relevance in later practical work: specifically both the Group Project this year and your individual project next year. Familiarity with and competent use of standard tools and techniques can make your work on these projects significantly more efficient, and all assessment of practical work are entitled to assume this fluency when judging whether the amount of work done was more or less than could be reasonably expected of you.

For this course I count the printed lecture notes as more important than the lectures themselves, and I also believe that your practical skills will only develop with practical experience, so I would urge you all to try using each of the tools and techniques mentioned here. I will generally only explain the *simple* ways of using each facility, and as you gain confidence it may be that you will benefit if you deepen your understanding by reading the `man` pages or other documentation. Many of the commands will display a concise 'reminder' of their usage by simply invoking them the argument `-?` or `--help`. "Info pages" can also be useful, and can be viewed in `emacs` by invoking `Ctrl-h i`.

There is at least an attempt to define a standardised set of Unix tools, including the shell: those interested in fine and authoritative details might like to follow up on the IEEE POSIX Shell and Tools specification (IEEE Working Group 1003.2). Note that IEEE does not publish its standards on the web or otherwise make the freely available: it supports its standardisation work through sales of both the standards and various certification services. More information about their activities can be found on the website `http://standards.ieee.ord/`.

A thread I hope will run through my presentation is that the tools discussed are not totally arbitrary in their design (despite some of the initial impressions that they give). There is at least a part of their construction that concerns itself with compatibility of ideas from one tool to the next and of exploitation of powerful and general computer science fundamentals such as regular expressions. The "Unix philosophy" that I mentioned earlier is that (ideally) the world would contain a number of tool components, each addressing just one problem. Each individual tool would then be small, easy to learn but completely general in its treatment of the limited class of problem that it addressed. The Unix approach is then to solve typically messy real-world problems by combining use of several such basic tools. In this spirit there will be a small number of major ideas underlying all the material covered here:

1. Complex tasks are often best solved by linking together several existing programs rather than be re-inventing every possible low-level detail of the wheel over again;

2. Regular expressions, seen in the Part IA course as a mathematical abstraction of pattern description, generalise to provide amazingly powerful and flexible (if sometimes obscure-looking) capabilities;

3. There should be a smooth transition between the tasks you perform one at a time interactively and those that need real programs written to perform them. The Unix tool tradition is particularly strong on helping to automate tasks of medium complexity;

4. Mouse/Menu interfaces are utterly admirable for making editors easy to use for the untutored or casual user. They are also helpful when your main concern is the visual appearance of a page of text, since they can make it easy to select a block of text and change its attributes. But for very many other tasks a keyboard based (vermin free?) approach can let a short sequence of keystrokes achieve what would otherwise require much mouse movement and the switching of concentration between mouse and keyboard. The learning effort required pays off as you are able to get your work done faster.

The topics covered here are somewhat inter-related and so a strictly linear and compartmentalised coverage would not be satisfactory. I will thus often refer back to concepts sketched earlier on and flesh out additional details in examples given later in the course. In four lectures it must be clear that I can not cover all of the facilities that Unix provides, and the language `perl` that I discuss towards the end could of itself justify a full-length lecture course and a host of associated practical classes. You must thus be aware that this course is going to be superficial, and those of you who already count yourselves as experts are liable to find many of your favourite idioms are not covered. However the lectures can (I hope) still form a good starting point for those who are relative Unix beginners, while these notes can be a reminder of what is available, a modest source of cook-book examples and a reminder of which parts of the full documentation you might want to read when you have some spare time.

3 The "Unix shell"

These first few sections will recapitulate on material that you will (mostly) have come across in the introductions to Unix you had at the start of the Part IA Java course, or that were mentioned in part of the Operating Systems thread. Repetition in these notes will help keep this course self-contained, although the lectures will skim over this information very rapidly. See [2] for a tolerably concise expansion of what I have included here.

Part of the Unix design involved making all the functionality supported by the operating system available as function calls, and making as many of these calls as possible available to all users. In other words a deliberate attempt was made to arrange that Unix security only needed a very small set of operations to be run as privileged system tasks. Partly as a demonstration of this, the system ensured that the *shell* could be written as an ordinary user-mode program. The shell comprises the fundamental interface the user sees to Unix: it is the component of Unix that lets you type in commands which it then executes for you. Many other operating systems give their shells private and special ways of talking to the inner parts of the operating system so that it is not reasonable for a user to implement a replacement.

Two consequences have arisen. The first is that there are *many* different Unix shells available. This can be a cause of significant confusion! The second is that good ideas originally implemented in one of these shells have eventually found their way (often in slightly different form) into the others. The result is that the major Unix shells now have a very substantial range of capabilities, and the way in which these are activated has benefited from a great deal of experimentation and field-testing. The original Unix shell is known as the Bourne Shell (after Steve Bourne, who after leaving Cambridge went to Bell Laboratories where his enthusiasm for Algol 68 had its effects). The major incompatible shell you may come across is the *C shell*, where the "C" is both to indicate that its syntax is inspired by C, and also (given a Unix tradition of horrible puns) because one expects to find shells on beaches, so a C-shell is an obvious thing to talk about. There have been many successors to these two shells. The one that you are (strongly) encouraged to use here is basically upwards compatible with the Bourne Shell, and is know as `bash`, the Bo(u)rn(e)-Again SHell. `bash` is part of the excellent GNU implementation of the Unix Tools, and is available on almost every platform imagineable. Anyone thinking of using a C-shell derivative is advised to read Tom Christiansen's article "Csh Programming Considered Harmful" [1].

Whenever you are typing in a command at the usual Unix command-prompt you are talking to your current shell. Also if you put some text in a file and set the file to have "executable" status (eg by saying `chmod +x filename`) then quoting the name of the file will get the shell to obey the sequence of commands contained. In such case it is considered standard and polite to make the first line of the file contain the incantation

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

where `/bin/sh` is the full file-name of the shell you are intending to use. For shell scripts like this it is generally safest to restrict yourself to the facilities and syntax of the basic Bourne shell, even if you use something more powerful when interacting. The "`#`" mark makes this initial line a comment, and the following "`!`" and the fact that it is the very first thing in the file mark it as a special comment used to indicate what should be used to process the file. Some of the examples given here will be most readily tested interactively while others will be best put into files while you perfect the lengthy and messy runes.

Some small and common things that you may have thought of as commands are in fact built into the shell (for instance `cd`[2]). But the most interesting shell features relate to ways to run other programs and provide them with parameters and input data.

4 Streams, redirection, pipes

Central to the Unix design is the idea of a stream. Streams are the foundation for input and output, and at one (fairly low) level they are identified by simple small integer identifiers. When a program is started the shell provides it with three standard streams, with numeric identities 0, 1 and 2. The first of these is *standard input*, and programs tend to read data from there if they have nothing better to do. The second is a place for *standard output* to be sent, while the third is intended for *error messages*. If you start a program without giving the shell more explicit information it will connect your keyboard so it provides data for the standard input, and it will direct both of the default output streams to your screen.

These standard streams can be redirected so that they either access the filing system or provide communication between pairs of programs. The importance (for today!) of file redirection is that it means that a program can be written so that it just reads from its standard input and writes to its standard output. Using redirection the shell can then cause it to take data

[2]For the purposes of this course I am going to suppose that some of the basic Unix commands are already familiar. But the suggested textbook will give brief explanations even if I do happen to mention something that you have not seen or that you have forgotten about.

from one file and write its results to another. The program itself does not have to bother with any file-name decoding etc.

```
my_program < input_data.file > output_data.file
```

If `>>` is used as a redirection operator the new data is appended to the output file. This can be very useful when executing a sequence of commands:

```
#!/bin/sh
echo "Test run starting" > log.file
date >> log.file
my_program >> log.file
echo "end of test" >> log.file
```

Two programs can be linked so that the (standard) output from the first is fed in as the (standard) input to the second. The fact that this is so very easy to arrange encourages a style where you collect a whole bunch of small utilities each of which performs just one simple task, and you then chain them together as *pipes* to perform some more elaborate process. I will use this in quite a number of the examples given later in these notes. One useful program to put in a pipe is `tee` which passes material straight from its input to output, but also diverts a copy to a log-file. The following (not very useful) example uses `cat` to copy an input file to its standard output. `tee` then captures a copy of this to a log file, and passes the data on to `my_program` for whatever processing is needed.

```
cat input.file | tee log.file | my_program
```

A further use of pipes and `tee` is as follows where the standard output from a test run is permitted to appear on the screen but a copy is also diverted to a log file in case detailed examination is called for at a later date. The output is piped through `more` to make it possible to read it all even when it is too long to fit on a single screen. The Unix enthusiast would point out the power of pipes where the functionalities of both `tee` and `more` are being combined without the need for a messy composite utility.

```
my_second_program | tee log.file | more
```

Especially when debugging code it is often important to be able to redirect the error output as well as the regular one. This is one of the areas where the exact syntax to be used depends on which shell you are using, and so my use of the Bourne Shell or one of its derivatives does matter. For such shells the form `2> error.file` redirects the standard error file (descriptor number 2) so that material is sent to the named file.

Its also sometimes useful to be able to redirect standard error to standard output, so you feed the combined stream into another program. Again, the exact syntax is shell specific, but the following works for `bash` and is quite useful when you're compiling a program with lots of errors:

```
make my_program 2>&1 | more
```

Likewise, in a shell script it sometime useful to be able to `echo` messages to standard error to standard error:

```
echo 'Arghh! It's all broken!' 1>&2
```

A final common redirection feature known as 'Here Documents' is activated using `<<`. This makes it possible to embed an input document within a shell script file. After the doubled angle bracket you put some word, and the standard input to the command activated will then be all lines form the command input source up to one that exactly matches this word:

```
#!/bin/sh
cat << XXX > output.file
line 1 to go in the new file
line 2 to go in the new file
XXX
```

5 Command-line expansion

When the shell is about to process a command it first performs some expansion on it. It will interpret some sequences of characters as patterns, and replace them with a list of all the names of files that match those patterns. As a special and perhaps common case the single character "`*`" is a pattern that matches the names of all[3] files in the current directory. For these purposes a sub-directory is just another file. Because this wild-card expansion is performed by the shell before a command is executed its effects are available whatever command you are using. Perhaps a convenient one to try is `echo` which just prints back its parameters to you:

```
echo *
```

will display a list of the files in the current directory. Of course to achieve *this* effect you would normally use `ls` which lays out the list neatly and provides lots of jolly options, but use of a pattern means that you can send the list of file-names to any program, not just to `echo`. For now the important components of a pattern are

1. Most characters stand literally for themselves;
2. An asterisk (`*`) matches an arbitrary string of characters. In file-name expansion file-names that start with a dot (`.`) are treated specially and the wild-card asterisk will not match that initial dot;

[3]Well all except for the "hidden" file-names that start with a dot...

3. A question mark (`?`) matches any single character, again except for an initial dot;

4. A backslash (`\`) causes the following character to lose any special meaning, and so if you need a pattern that matches against an asterisk or question mark (or indeed a backslash) one may be called for;

5. Quotation marks (either single or double) can also be used to protect special characters. Note that as well as `*` and `?` the Unix shell may be treating `>` and all sort of other punctuation marks specially, so in case of doubt use quotation marks or backslash escapes fairly liberally!

In addition to file-name expansion the shell expands commands by permitting reference to *environment variables*. This is indicated by writing a variable name preceded by a dollar sign (`$`). It is also legal to write a variable reference as `${name}` where the braces provide a clear way of indicating where the variable name ends. There are liable to be quite a few variables predefined for you, some set up by the shell itself, some by scripts that are run for you when you log on. There are in fact two subtly different classes of shell variable and anybody who is going to get involved in serious shell programming would need to understand the differences (which mostly relate to visibility from one shell session or script to another). In this course I am suggesting the use of `perl` for any elaborate scripting so I can afford to omit this explanation.

To set a new variable you may write

```
variable_name=value ; export variable_name
```

where the use of `export` is not always necessary but at the level of detail I am including here I suggest that you always include it. `bash` also provides a non-standard shorthand form of the above:

```
export variable_name=value
```

A slightly more concrete example shows that with a lot of shell variables set up the actual commands that you issue may turn out to be almost entirely built out of references to variables.

```
LANGUAGE=java;       export LANGUAGE
COMPILER=javac;      export COMPILER
OPTIONS=;            export OPTIONS
SOURCE=hello_world; export SOURCE
$COMPILER $OPTIONS $SOURCE.$LANGUAGE
```

Especially in script files there is a great deal to be said for establishing variables to hold the names of compilers that you use and of the options that must be passed to them, since it makes everything much easier to alter if you move your programs to a slightly different environment later on. For instance on various Unix machines that I have used the C compiler is called sometimes `cc`, sometimes `gcc`, and sometimes `c89`, maybe `ncc` and even `/opt/EA/SUNWspro/bin/cc`! Changing just one setting of a variable to allow for this is neater than making extensive edits throughout long scripts[4].

A rather different but convenient use of variables is to hold the names of directories. If you often work within a directory with a rather long name, say `/home/acn1/Project/version-1_0_3/source`, then you might set a variable (say `SRC`) to that long string. Then you can use `$SRC` freely on the command line to allow you to select your important directory or refer to files within it with much less typing that you might otherwise need. An additional advantage of this strategy is that when you move on to version `1_0_4` you can just change the one place where you define this variable and now you will naturally access the newer location. You can see the set of environment variables that are currently defined in a shell using the `printenv` command.

Another key-stroke saving tip is to define shell functions that alias other commands (or sequences of commands). For example:

```
function m () { more }
function ll () { ls -al }
```

The same effect could be achieved by creating shell scripts and placing them on the search `PATH`, but defining a function is more efficient.

Within a shell script `$1`, `$2`, ... refer to arguments passed when the script was started, `$*` expands to a list of all the arguments, while `$#` is replaced by the number of arguments provided. Consider a file called `demo` that contains

```
#!/bin/sh
echo Start of $0, called with $# arguments
echo All args: $*
echo Arg 1 = $1
echo Arg 2 = $2
# Let's put another comment here.
```

which also illustrated that "argument zero" will be taken to refer to the name of the script that is being executed. A use of the above script might be

[4]In a DOS/Windows world, of course, the C compilers I use or have used are called wcl386, cl, ztc, gcc (maybe 3 versions!), and bcc32. And they all need different sets of options and parameters, so the confusion is not unique to Unix.

```
demo hah                   | what I typed in
Start ./demo with 1 args | output from script
All args: hah
Arg 1 = hah
Arg 2 =                    | $2 => empty string
```

Redirection allows one to send the output from a command to the standard input of another. Sometimes you may want to incorporate the (standard) output from one command as part of another command. This is achieved by writing the first command within back-quotes (‘). A sensible example of this in use will be shown later on, but for now I will illustrate it with

```
echo Today is ‘date‘
```

6 `find` and `grep`

One of the expectations that comes with Unix is that it should be easy to specify that operations should be performed upon multiple files by issuing just one command. File-name expansion as described earlier provides the simplest way of listing a bunch of files to be processed: sometimes it is useful to have rather more subtle selection and filtering procedures. The tool `find` is used when this should be based on the file's name and attributes (eg date of last update), while `grep` inspects the contents of files.

6.1 `find`

The `find` command takes two groups of arguments. The first few arguments must be path-names (ie typically the names of directories), and the subsequent ones are conditions to apply when searching through the directories mentioned. The conditions that can be used include tests on the name, creation and modification date, access permissions and owner of files. A special "condition" `-print` causes the name of the file currently being processed[5] to be sent to the standard output, and it will often be shown as the final item on the command-line. Unless combined using `-o` which stands for *OR* all previous conditions must be satisfied if the `-print` is to be activated. Conditions may be negated using `!`.

The conditions `-atime`, `-mtime` and `-ctime` test the access, modification or creation times of files. They are followed by an integer. If written unsigned

[5]Some implementations of `find` have an implicit `-print` if no other action is specified, but do not rely on that even if the one that you usually use does it

they accept files exactly that many days old. If it is written with a + sign they accept files older than that, and a - asks for files younger. A condition -name is followed by a pattern much like those previously seen in file-name expansion, and checks the name of the file. You will normally need to put a backslash before any special characters in the pattern. There are lots of other options, including ones to execute arbitrary programs whenever a file is accepted, but those of you who want to use them can read the full documentation.

Plausible uses are illustrated in the following examples:

1. List all files that have not been accessed for at least 150 days. These are obvious candidates for moving to an archive, or compressing or even deleting! Note that by searching in the current directory (.) even files whose names start with a dot will be listed here.

   ```
   find . -atime +150 -print
   ```

2. List files that have been created during the last week. File-name expansion means that the * is turned into a list of all files in the current directory apart from those whose name starts with dot. Reminding oneself of recently created or modified files may be useful when you want to consider what to back up.

   ```
   find * -ctime -7 -print
   ```

3. Delete all files in the current directory or any sub-directory thereof if their name end in ".old". The -i flag to the rm command gets it to ask the user for confirmation in each case, which makes this command a little safer to issue. Observe the back-quotes to get the output from find presented as command-line arguments to the rm command.

   ```
   rm -i `find . -name \*.old -print`
   ```

4. Search the current directory and subdirectories for all entries that are *files* (excluding directories), and that have the 'other user' execute permission bit set.

   ```
   find . -type f -a -perm -001 -print
   ```

   Alternatively the xargs command could be used to build the call to rm. This command builds a command from an initial command name together with whater if finds on its standard input, and works better than the previous scheme when there are a very large number of arguments to be passed[6]:

```
find . -name \*.old -print | xargs rm -i
```

5. List any files in my `program` directory that are empty and whose name does not start with a `tmp`.

```
find program -size 0 ! -name tmp\* -print
```

`find` is obviously valuable as an interactive tool, perhaps especially for helping keep your file-space tidy. It is also a valuable building-block in scripts.

6.2 grep

`grep` is the first tool that I will describe here that makes serious use of the Unix interpretation of regular expressions. Its use is

```
grep <options> <regular-expression> <file(s)>
```

where the options may include `-i` to make searches case insensitive or `-c` to make it just count the number of matches found in each file. Normally `grep` searches through all the files indicated and displays each line that contains a match for the given regular expression. The option `-l` gets you just a list of the names of files within which there are matches. The sense of the matching can be inverted with the `-v` option.

You need to be aware that there are related command called `egrep` and `fgrep` that support different degrees of generality in the pattern matching. Furthermore on some computers you will find that the program invoked by the `grep` command has either more or less capability than is mentioned here. This all arises because matching against very general regular expressions can be an extremely expensive process so the early Unix tool-builders decided to provide three different search engines for trivial (`fgrep`), typical (`grep`) and ambitious (`egrep`) uses. In describing the regular expression formats that are available I will mark ones that need `egrep` with an (†). Given that todays computers are pretty fast you might like to standardise on using `egrep` to reduce your worry on this front.

[6]Specifically when you issue an ordinary command, including via the backquote construction, there may be a limit on the lenth of the command-line that can be handled. `xargs` goes to some trouble to invoke programs properly even when they are to be passed utterly huge numbers of arguments.

All the real interest and cleverness with `grep` comes in the regular expressions that it uses. You will recall the rather spartan definition of a regular expression used in the Part IA course that introduced them. Those provided everything that was actually *needed* to describe any regular language, but in many realistic cases there is a very great benefit in using additional short-cuts. The following are the more important of the constructs supported by `grep`, and as we will see later most of them are also used with `sed` and `perl` as well as various other Unix-inspired tools.

a,b,. . . In general characters in a regular expression stand for themselves. If one of the special characters mentioned below is needed as an ordinary literal that can be arranged by sticking a backslash in front of it. Note then (of course) that to get this backslash through to where `grep` will find it you may need either quote marks or a yet further backslash, and things can start to look messy!

A B Concatenating regular expressions works in the obvious manner. An effect is that strings of literal characters can be given and match words in much the way you might expect;

`(` **A** `)` (†) Where necessary you may use parentheses to group sub-parts of a complicated expression;

A `|` **B** (†) Alternation is written using a vertical bar, which may be read as *OR*;

A`*` The star operator applies to the previous character or bracketed expression, and matches zero or more instances of it;

A`+` (†) Much like the star operator, but accepts one or more instances of things that match the given pattern;

A`?` (†) Zero or one matches for the given item;

A`\{`*n*,*m*`\}` From n to m repetitions. Amazingly this construct is only guaranteed to be available in `grep` and for `egrep` you may be able to achieve the same effect with a pattern that omits the backslashes. This is a natural generalisation of the more common cases that use `*`, `+` and `?`.

`[`**a**`-`**z**`]` This matches a single character, which must be one of the ones listed within the brackets. Ranges of characters are shown with a hyphen. If you put a hyphen or close (square) bracket as the very first character then it is treated as a literal, not as part of the syntax of the construct. The mark "^" can be used at the start of a pattern to negate the sense of a match;

. A dot matches any single character except a newline. Thus `.*` matches any string of characters not including newlines;

^ and $ Normally patterns are looked for anywhere within a source line. If you put a ^ at the start of an expression it will only match at the start of a line, while a `$` at the end ensures that matches are only accepted at the end of a line. Use both if you want to match a whole line exactly;

\< and \> These allow you to insist that a certain place within your pattern matches the start or end of a word. This facility is only supported in some implementations of `grep`.

Again I think that the possibilities are best explored via some examples. Firstly I will give just regular expressions, and then I will build them into complete commands showing `grep` in a potentially useful context:

1. A pattern that matches words that start with a capital letter but where the rest of the characters (if any) are lower case letters and digits or underscores

   ```
   [A-Z][a-z0-9_]+
   ```

2. The string "`#include` at the start of a line, apart from possible leading blanks

   ```
   ^ *#include
   ```

3. A line consisting of just the single word `END`

   ```
   ^END$
   ```

4. A line with at least two equals signs on it with at least one character between them

   ```
   =.+=
   ```

5. Find which file (and which line within it) the string `class LostIt` is in, given that it is either in the current directory or in one called `extras`

   ```
   grep 'class LostIt' *.java extras/*.java
   ```

6. Count the number of lines on which the word `if` occurs in each file whose name is of the form `*.txt`.

   ```
   grep -c "\<if\>" *.txt
   ```

   The output in this case is a list showing each file-name, followed by a colon and then the count of the number of lines which contain the given string. The use of `\<..\>` means that `if` embedded within a longer word will not be recognised.

7. As above, but then use `grep` again on the output to select out the line that end with `:0`, ie those which give the names of files that do not contain the word `if`. This also illustrates that if no files are specified `grep` scans the standard input.

   ```
   grep -c "\<if\>" *.txt | grep :0\$
   ```

8. Start the editor passing it the names of all your source files that mention `some_variable`, presumably because you want to review or change just those ones. You could obviously use the same sort of construct to print out just those files, or perform any other plausible operation on them.

   ```
   emacs `grep -l some_variable *.java`
   ```

   Note that `grep` has its own idea of what a "word" is, and so in some circumstances you may want to write a more elaborate pattern to cope with different syntax.

Regular expressions do not provide a sufficiently general way of describing patterns to allow you to do real parsing of programming languages, and as present in `grep` they do not even make it easy to distinguish between the body of your program, comments and the contents of strings. However with a modest amount of ingenuity they can often let you specify things well enough that you can search for particular constructions that are interesting to you. Some people may even go to the extreme of laying out their code in stylised manners to make `grep` searches easier to conduct!

7 Exit codes and conditional execution

When commands terminate they exit with an integer value known as a *exit code*. In general, non-zero exit codes indicate that some sort of error or other abnormal condition occurred. The exit code of the most recently executed foreground command can be accessed via the `$?` environment variable. For example:

```
echo "numbers" | egrep '[0-9]+' >/dev/null ; echo $?
1
echo "1234567" | egrep '[0-9]+' >/dev/null ; echo $?
0
```

It is possible to make the execution (or not) of commands dependent on the exit code of previous commands using the `||` and `&&` operators. These operators provide short-circuiting OR and AND functions respectively. In the case of `A || B`, `B` is executed iff `A` has a non-zero exit stats. Conversely, for `A && B`, `B` is executed iff `A` has a zero exit code. For example:

```
echo "1234567" | egrep '[0-9]+' >/dev/null && echo "numbers"

ping -c 1 bailey || ping -c 1 lundy || echo "network broken?"
```

More complex conditional execution can be achieved using the `if/then/else`, `for`, `while` and `case` statements of most shells. These statements are frequently used in conjunction with the `test` command, for which the abbreviation '`[`' is frequently used[7]. `test` can be used to check the existence, access permissions, and modification times of files, as well to perform comparisons between pairs of strings and even integers. For example:

```
if [ -e foo -a -e bar -o $1 = "skip" ]
then
   echo "files foo and bar both exist, or arg1 == skip"
fi

while [ ! -r foo ]
   do  echo "foo is not readable" ; sleep 1 ; done

case `arch` in
*hpux*)
  PSOPT="-eafl"  # HPUX requires strange options to 'ps'
  ;;
*)
  PSOPT="auxw"   # These options are OK for Linux, OSF etc
  ;;
esac
```

When using these commands it can be tricky to remember where it is necessary (or forbidden) to insert command terminators, such as the newline character or '`;`'. Perhaps, the easiest way to learn more about these

[7] On many older Unix systems, `/usr/bin/[` was a filesystem soft-link to `test`. I've heard at least one apocryphal story of an over zealous sysadmins tidying up `/usr/bin/` by deleting 'spurious' files such as `[`...

commands is by examining other people shells scripts. A good source is to look in a Unix system's boot scripts directory. On most Linux boxes this is `/etc/rc.d/init.d`.

8 Shell script examples

The following section contains a number of fragments from shell scripts that I find useful. I make no warranty as to their fitness for the purpose intended, or even that they demonstrate good programming style.

The following is handy for searching a directory hierarchy of 'C' source files looking for a particular identifier. Since all command line arguments are passed in to `grep`, it's possible to ask it to e.g. ignore case using the `-i` option.

```
function trawl () {
   find . \( -name '*.[chsS]' -o -name '*.[ch]?' \) \
     -print | xargs fgrep -n $*
}
```

This function can be used to send a kill signal to the named process(es):

```
function killproc() {
   pid=`/usr/bin/ps -e |
   /usr/bin/grep $1 |
   /usr/bin/sed -e 's/^  *//' -e 's/ .*//'`
   case "$pid" in [1-9]*) kill -TERM $pid;; esac
}
```

The following function can be used to add an entry to the tail of the PATH string, or deletes the entry if it's already present.

```
function addpath () {
   case $PATH in
   *${1}*)         PATH=`echo $PATH | sed "s+:${1}++"` ;;
   *)              PATH="$PATH:$1" ;;
   esac
   echo PATH=$PATH
}
```

9 `make` and project building

When building a serious program you will have a number of different source files and a collection of more or less elaborate commands that compile them all and link the resulting fragments together. For large projects you may have helper programs that get run to generate either data files or even fragments of your code. When you have edited one source file you can of course re-build absolutely everything, but that is obviously clumsy and inefficient. `make` provides facilities so you can document which binary files depend on which sources so that by comparing file date-stamps it can issue a minimal number of commands to bring your project up to date.

The information needed has two major components. The first is a catalogue of which files depend on which other ones. The second is a set of commands that can be executed to rebuild files when the things that they depend upon are found to have changed. By default the utility looks for this information in a file called `Makefile`[8]. In a practical Makefile there will often be a substantial amount of common material used to make the actual rules themselves more compact or easier to maintain. In particular variables will often be used to specify the names of the compilers used and all sorts of other options. My first sample (or template) Makefile will be for use with an imaginary programming language called `frog`. It imagines that source files are first compiled into object code, and then linked to form the final application.

```
# Makefile for "princess" program

COMPILE = frogc
OPTS    = -optimise -avoid_lillypads=yes
LINK    = froglinker

princess: crown.o tadpole.o
        $(LINK) crown.o tadpole.o -to princess

crown.o:  crown.frog
        $(COMPILE) $(OPTS) crown.frog

tadpole.o: tadpole.frog
        $(COMPILE) $(OPTS) tadpole.frog

test.log:  princess test.data
        date > test.log
        princess < test.data >> test.log
```

[8]You can also use `makefile` without a capital. Many Unix users (slightly) prefer the capitalised version because it results in the file being shown early on in the output from `ls` when they inspect the contents of a directory.

```
# end of Makefile
```

The above file starts with a comment. Each line that begins with `#` is comment. Next it defined three variables, which are supposed to be the name of the compiler, options to pass to the compiler and the name of the linker. Separating these off in this way and then referring to them symbolically makes things a lot easier when you want to change things, which in the long run you undoubtedly will. Note the use of curly braces rather than round braces in Makefile variables.

The next few blocks are the key components of the file. Each starts with a line that has a target file-name followed by a colon, and then a list of the files upon which it depends. Following that can be a sequence of commands that should be obeyed to bring the target up to date. These commands must be inset using a tab character (n.b. not spaces). A blank line marks the end of such a sequence of commands. More or less anywhere it is possible to refer to variables, and using a dollar sign you can refer to either something defined in the `Makefile` itself or to a variable set up by the shell. Additional variable definitions can be passed down when `make` is invoked.

To use this you just issue a command such as `make test.log`, where you specify one of the declared targets. `make` works out how many of the commands need to be executed and so in the above case if nothing at all had been pre-built it would execute the commands

```
frogc -optimise -avoid_lillypads=yes crown.frog
frogc -optimise -avoid_lillypads=yes tadpole.frog
froglinker crown.o tadpole.o -to princess
date > test.log
princess < test.data >> test.log
```

If you do not tell `make` what to do it updates whatever target is mentioned first in your `Makefile`.

A true Unix enthusiast will feel that the above `Makefile` is too easy to read and that it does not include enough cryptic sequences of punctuation marks. A slightly better criticism is that as the number of source files for our princess increases the contents of the file will become repetitive: it might be nice to be able to write the compilation command sequence just once. This is (of course) possible. In fact there will usually be a whole host of built-in rules and predefined variables (they are typically called *macros* in this context) that know about a wide range of languages, and the most you will ever want to do will be minor customisation on them. To illustrate the power of `make` I will stick with my imaginary Frog language. To tell `make` a general rule for making `.o` files from `.frog` ones you include something like the following in your `Makefile`:

```
.frog.o:
        $(COMPILE) $(OPTS) $<
```

where the `$<` is a macro that expands to the name of the source file that needed recompilation. There are other slightly cryptic macros that can be used in rules like this. These funny automatically defined macros are needed so that you can refer to the files that the general file-suffix-based rule is being used on.

`$@` expands to the name of the current target, ie the file that is to be re-created;

`$<` expands to the name of the "prerequisite" file, ie the source file that had been seen to have a newer time-stamp than the target;

`$*` is like `$<` except that what it expands to does not include the file suffix.

By default `make` stops if one of the commands it tries to run fails, and it then deletes the associated target. The idea here is that if just one of your source files contains a syntax error then everything will be re-built up to the stage that that is detected, and things will be left so that a subsequent invocation of `make` will try that file again and then continue.

There are in fact a few further thing that I ought to mention with regard to `make`: if you use file suffixes other than the ones that are initially known about you may need to declare them and specify their ordering. In the case being discussed here it would be necessary to specify first an empty list of suffixes (to cancel the built-in list[9]) and then list the ones that are desired. The various file suffixes should be listed in order, with generated files first and original source ones last:

```
.SUFFIXES:
.SUFFIXES: .o .frog
```

It is also recommended that you put a line that says

```
SHELL = /bin/sh
```

in every `Makefile` so that even if it is invoked by somebody who is using a non-standard shell its internal command processing will behave in a standard manner.

Again (as you might expect) there are other declarations that can be provided for various specialist uses. I will not even mention them here.

[9] At least some version of `make` appear to require this.

Some versions of make provide extra facilities, notably the opportunity to build conditions into the file so that different things happen based on the values of macros. Another extension is the ability to reference other files so that it is as if their contents had formed part of the original Makefile. I suggest that you avoid any such features even when they do make life a lot easier, at least until you have had significant experience moving programs from one computer to another. Some people would disagree with me here, perhaps suggesting that whenever you move to a new computer you should fetch and install a copy of the GNU version of make on it so you can be certain that all of its capabilities are available. I will re-phrase my advice to suggest that you stick with *very* plain and simple Makefiles at least until you feel comfortable re-building GNU make from source and installing it on new computers!

It is well worth using a Makefile as the repository for many more commands than just those to recompile your code. You can usefully put in a target that tidies up by deleting all object and executable files (leaving just the original sources present), ones to run test cases, commands for formatting and printing the manual, scripts that pack up a version of your program for distribution and interfaces to whatever backup/archive discipline you adhere to. The one file can then end up as documentation of all the major procedures associated with the management of your program: it is perhaps sensible then to make sure it has plenty of informative comments in it.

It is perhaps at this stage worth noting the command touch that resets the date on a file to make it look as if it is new. Use of this can sometimes allow you to trick make into assuming that some binary files are new enough that they do not need re-building even though the general rules given suggest otherwise. This can be helpful if you make changes in some source files that you are certain do not really call for re-compilation: eg correction of spelling errors in their comments.

The following example is a simple Makefile for a collection of C sources. It uses the makedepend utility to auto-generate the dependencies list and append it to the end of the Makefile. Generating the dependencies in this way avoids obscure bugs that can be caused when hand-entered dependencies are inaccurate and result in some files failing to be compiled when certain #include'd files are updated.

```
TARGET   = my_prog
SRCS     = a.c b.c c.c
OBJS     = $(SRCS:.c=.o)

CC       = gcc
CFLAGS   = -O2
```

```
INCLUDES = -I.
LD       = ld
LDFLAGS  = -Bdynamic
LIBS     = -lm
RM       = rm -f

.c.o:
	$(CC) $(CFLAGS) $(INCLUDES) -c $<

$(TARGET): $(OBJS)
	$(LD) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

clean:
	$(RM) *.o $(TARGET)

depend:
	makedepend $(INCLUDES) $(SRCS)

# DO NOT DELETE THIS LINE -- make depend depends on it.

a.o: magic.h magic2.h
b.o: magic.h magic2.h
c.o: magic2.h
```

10 rcs and friends

The most common cause of corrupted or lost files these days is not liable to be hardware failure, viruses or rogue software. It will be carelessness on the part of the owner of the files. The proper protection against ill-conceived editing and false-starts towards program upgrades are best based on keeping a fairly detailed incremental record of changes made to all files. Because most changes are rather small these can be stored quite compactly by keeping a base version of each file and a list of changes made to it. The program `diff` which is discussed later on can compare two versions of a file to generate just such a list of changes. If properly organised such a scheme could have just one file representing a base version of a module, the most recently released fully-tested version and several experimental versions. Any one of the versions stored could be re-created by applying the relevant set of stored edits to the base version of the file. Having got that far it would seem natural to attach commentary to each set of updates to document their author and intent, and to accept the fact that several programmers might be working on just one project, and all of them might be making their own separate changes. With all this in place editing a file should *never* destroy information, because it will always be possible to reconstruct the state the file was in before it. It is still necessary to back up files to protect against system failure or wholesale deletion of the code database, but overall everything should feel a lot safer. What I have just described is a *revision control system*, and the Unix tool `rcs` is the prime example[10]. Note that `rcs` is at least as relevant when developing documentation (or indeed literary works) as when writing programs.

Getting started with `rcs` is easy. Change directories into the one where your source files live. Create a sub-directory called `RCS` which will be where the system keeps its database. You then *never* mess about inside that directory directly, and periodically copy its contents to floppy disc, magnetic tape or some other fully secure and for choice remotely stored medium. Now suppose you have a set of files that make up the base-line version of your program: imagine they are called `Arthur.java` and `Norman.java`. You issue the commands:

```
ci Arthur.java
ci Norman.java
```

and when you have finished typing in the description of the two files that you are asked for and you look at what has happened you find that your files seem to have vanished. In fact they have been entered into the database with a "revision number" set to 1.1.

`ci` stands for *check-in*. Given that it is perhaps not too much of a surprise that the converse command is called `co` for *check out*. This has two rather different uses. The first is when you just want to read the file (and the most recent version of it at that). Then you say just

```
co Arthur.java
```

and you should find that the file has re-appeared. You can print it or compile it. If you want to edit it you ought to say

```
co -l Arthur.java
```

where the `-l` stands for *lock*. Part of what `rcs` does for you is to arrange that only one user can lock a file at any one time, so if you are working in a multi-user project[11] and this succeeds then you know that none of your colleagues will be able to lock that particular file until you have finished making your edits and have checked it back in again. If you specify the lock

[10] `sccs` is another, but at least locally `rcs` is distinctly preferred.

[11] Such as the Lent Term group project...

flag (`-l`) with `ci` it checks the file in but leaves a copy outside the database for you to continue editing.

Note that with this scheme you may need to have a suitably elaborate `Makefile` that checks out (without locking) all the files that it needs to compile. Unlocked checking out is possible at any time. All that locking does is to ensure that the only person subsequently allowed to check a file back in is the person who successfully obtained a lock for it.

The command `rcsdiff` compares the version of a file that you have checked out with the version in the database so you can get a quick reminder of what you have just changed.

Each time a file is checked in it is given a new revision number. These numbers normally run 1.1, 1.2, 1.3, ... You can check out a specific one of these by saying (eg)

```
co -r1.4 Norman.java
```

While checking in a file `rcs` scans it for the string `$Id$` and if it finds that it replaces it with a longer string that gives the name of the file, its author, revision number and the date. This is a textual substitution so you should normally place the marker within a comment or a string. If you put it in a string you may be able to access this version information when your program is run.

If the string `$Log$` is present in your source file the explanations you type in when `ci` prompts you are collected there, (almost) painlessly accumulating update history as part of your source files. Some other similar substitutions are made for other words enclosed in pairs of dollar signs.

There are (of course) rather more facilities and options than just that, and in particular special provision is made for the case where a project is totally under the control of just one programmer and so the full discipline of locking is too heavy handed. The only option I will mention here is `co -d`*date* where the date can be specified in almost any even half-plausible syntax (the only trick is that the suffix `LT` is often useful, and indicates that the clock should be measured in local time rather than (say) the time zone relevant for California or Japan. In this country you may like to use `GMT` instead, and allow for summer time yourself? This option recovers the latest version of the file whose date is no later than the one given. Eg

```
co "-d monday" file1.xxx
co "-d 2 july" file2.yyy
co "-d 8:00 pm gmt" file3.zzz
```

For the schemes that permit branches in the tree of versions see the manual pages: the need for that should not arise until you have had time to get thoroughly comfortable with the basic commands. Another command worthy of investigation is `rcsfreeze`. It enables you to assign a symbolic revision name to a *set* of source files. You might wish to do this from time to time to denote when your set of source files are in a state that you may wish to return to, for example, at landmark points in the development cycle.

Especially for projects that involve collaboration between programmers who do not share a common computer or who are otherwise mildly remote from one another it may make sense to investigate `cvs` or the new GNU `prcs` system. This is not available on all Unix systems but where it is it provides an alternative interface to the facilities of `rcs` which may prove more convenient.

One issue that has not been addressed so far is that of what to do if you fail to obtain a lock on a file that you want to change. The easy answer is to contact the person who does have a lock and see how long they expect to be working with that file. In some cases you may instead choose to check the file out unlocked and edit it locally. Then when (later on) you do succeed in locking the file you can try to merge your own edits with the changes that the other programmer made. `cvs` goes to some trouble to help with this, and if you and the other programmer made sufficiently well separated changes to the file it may be able to propose a good merged version for you.

Observe in any case that it is anti-social to lock a file for longer than you have to. It will generally count as good practice to plan your changes first, and then lock the file, make them and check it back in so as to keep the central tree free for as much of the time as possible.

On starting to use `rcs` you are liable to find two general frustrations. The first is that it is not integrated with or especially comfortable to use with `make` unless you fill your `Makefile` with lots of ugly `co` commands and then extra commands to delete the source files after you have compiled them. The second is that it stores its database in a rather rigidly defined location (ie the subdirectory `RCS` of the currently selected directory), and this means that all of the group of users working on a project tend to need to work in this one directory. To a small extent the option `co -p` which checks out a file but sends its contents to the standard output (whence it can be re-directed to any file anywhere) can be useful in scripts that allow for this. Another plausible strategy is to have a single common directory that all members of a group use *just* for `rcs`, and then each group members keeps their own private copy of all the files that make up the project. Files can then be copied to and from the common repository as need be (but without automatic protection against the possibility of two users performing clashing operations). Where available `cvs` provides distinctly better support for co-operative working.

11 Job control

The features mentioned here are probably ones that you are already familiar with, but for completeness I will mention them again.

Under Unix you can have several tasks running at once: one foreground task that you are interacting with and many background ones. Some background jobs will be active and running while others may be suspended. The command `jobs` displays information about the current situation, while `ps` can give information about tasks at a finer grain (eg when one job that you have launched in fact ends up generating a number of sub-tasks).

To start some task in the background you append an ampersand (`&`) to the command that starts it. Alternatively you can start the task as usual and then interrupt it using control-Z. This suspends its work. A suspended job can be allowed to proceed in the background using `bg` and any job can be made into the one that is directly connected to your terminal using `fg`. The jobs you have to work with may be referred to as %1, %2, ..., and the `jobs` command lists them so you can be reminded of the numbers allocated to each particular activity. To cancel a job you can use `kill` on it.

Ampersand can be used within shell scripts to launch tasks in the background. `$!` is assigned to the process ID (PID) of the most recently executed background command. The shell builtin function `wait` can be used to later wait for completion of the a background process, and collect its exit code. For example:

```
ping -c 1 bailey & pid1=$!
ping -c 1 lundy || wait pid1 || echo "Net broken?"
```

Especially on multi-user systems it is polite to prefix long and non-urgent commands with `nice`, which gets them run at a lowish priority thereby interfering less with other interactive work. Putting the word `time` before a command records the CPU and elapsed time it takes to execute it, splitting the computation time into that used directly by the task and that consumed by way of system overhead. It is unreasonable to expect timings to be accurately repeatable, if only because of effects of multi-tasking, virtual memory and caches.

```
nice make &    # make in background at low priority
fg             # bring it to the foreground
^Z             # suspend it...
bg             # .. and put back into background
time program < test.data > test.output # test time
jobs           # see what jobs I have active
kill %3        # kill job number 3
```

The disposition of background tasks that have not completed at the time that you log out and other such oddities can also be controlled, but such matters fall beyond the scope of this introduction (see `nohup`).

Having multiple background jobs running and controlled in this manner is probably less common than it used to be: nowadays many people will use Unix via X-windows and if they have several tasks that they want performed they will run them each in a separate window. This leaves the basic Unix job-control mechanisms to be used for starting new X windows and applications, as in

```
xterm -fn 10x20 -sb -sl 300 &
xclock &
```

where the `xterm` command launches a new command window (using a big font and providing 300 lines of scroll-back), while the second command pops up a clock for you. In each case the ampersand is important so that the new task runs concurrently with the shell from which it has been invoked.

Creating totally excessive numbers of Unix tasks can overload and ultimately crash the system and so is discouraged. Trying to speed up compilation by starting many jobs at the same time (one for each input file) is not in general useful: the jobs will compete for memory and disc access and can easily slow things down. However, if your using a multiprocessor machine, or compiling in an NFS mounted directory than there can be significant benefits. Investigate use of `make -j <n>`, but take care not to upset your colleagues on multiuser machines.

These days British University use of international networks is charged for, but transfers during the middle of the night are (at present) free[12]. Some large tests might be better run at times when a computer is not otherwise loaded. The command `at` allows you to schedule something to be executed later. Suppose that you have a program you would like recompiled overnight, and a set of commands stored in `script.file` that you want executed later on you could issue the commands:

```
at -f script.file noon tomorrow
at 02:00am
   cd /home/acn1/project
   make
   ^D
```

The first form specifies the commands to be obeyed by indicating a file that they live in. The second will prompt you to type in commands (I have

[12]The free period is 1am-6am. The charge is 2p per Megabyte at other times, which adds up to hundreds of thousands of pounds per year. Beware!

displayed them indented), and you end the list by typing control-D (ie the commands terminate at an end-of-file mark). As with `rcs` the format for dates and times that is supported is astonishingly flexible and general. The command `atq` should show you all the jobs you still have pending, but with luck you will not need to use it since `at` generally send e-mail to you when your task runs.

12 More shell facilities: history

If you use `bash` as your shell (as distinct from the original Bourne Shell) it keeps track of all the commands that you have issued and makes it easy for you to activate them again. This is achieved by making text that starts with an exclamation mark(`!`) expand into some variation on a previous command (much as things that start with a dollar get replaced by the value of a variable). The easiest cases arise when you want to repeat a command exactly as it was. Then the only issue is how to indicate which previous command is to be re-issued. There are five recipes:

1. `!!` just repeats the most recent command, whatever it had been;

2. `!-`*n* generalises on the above and repeats the command n back. The simple case `!!` is in fact an abbreviation for `!-1`;

3. `!`*n* re-plays command number n where n starts counting at the start of your session. The command `history` displays the commands that are stored along with their numbers in case you are not good at counting;

4. `!`*string* is a search. When you follow the exclamation mark with a string the shell searches back for the most recent command that starts with this string, and uses that one. This may well be the most generally useful variant;

5. `!?`*string*`?` is also a search, but by enclosing the string win question marks you indicate that the string should be looked for anywhere within the stored commands, not just at the start of the line.

When you have identified an old command with one of these recipes you can change the command. After all you may well be wanting to re-play it because you did not get it quite right last time. You can put an edit request after a history reference, and this is of the form `:s/`*pattern*`/`*replacement*`/`

where it is legal to use almost any character as the delimiter where I have used "/". As might be imagined this makes the indicated change to the old command. A special syntax is available if you want to make an edit to the most recent command. For instance if you had just typed in the mis-spelt command `histery` and wanted to correct it you could issue either of the following equivalent forms

```
!! :s/e/o/
^e^o^
```

and it is clear that the short-hand using `^` saves a useful amount of typing.

Because history substitutions are a part of the general shell command-line expansion process (along with file-name wild cards and variable references) you can put other text before or after a history reference again allowing you to get a modified version of a previous command. You can even make several history references on one line if that is at all useful.

The history mechanism in `bash` also responds to the arrow keys so you can browse your history that way. Control-R initiates a search through the history list. Personally, I find these interactive techniques much easier to use than the `!` mechanism, where you're never quite sure what its actually going to be executed...

One model that has been explored is to view terminal input as just another (slightly special) document that should respond to the keystrokes that you expect your editor (specifically `emacs`) to know about.

As well as history, `bash` also supports quite a sophisticated command line completion mechanism. Once enabled, it can be activated to suggest completions to a command line hitting the `TAB`, `ESC` or `Ctrl-]` keys. Various options control which databases it uses for completion; filename, history and host name completion are possible. Some other shells, such as `mash`, are even able to suggest corrections to minor spelling mistakes made on command lines!

13 Data tools (1): `tr`

There are many editing-style tasks that are slightly beyond the (easy use of) simple substitution capabilities of text editors but not complicated enough to make you want to rush out and write a new program to perform them. Unix provides several tools that cover common conversion tasks that you may come across. The first of these performs simple character substitution, and it is called `tr`. You should note that the version of `tr` that I describe here is the one documented in the book[2] and is as defined for Unix systems that follow the "System V" tradition. The version of `tr` on Unix systems descended from the BSD family differs, and the GNU `tr` as provided on Linux documents itself as not being *fully* compatible with either tradition. So before you use this command on any particular computer please check the `man` pages!

You give `tr` some simple options and two strings. It reads from its standard input and writes to the standard output, so it is usually seen with file redirection or pipes. Its basic use is to replace each character that is present in its first string with the corresponding member of the second. Strings can be abbreviated (as with `grep`) to show ranges of characters. Two plausible uses follow:

```
tr '[A-Z]' '[a-z]' < original.file > lowercase.file
cat message | tr '[a-z][A-Z]' '[b-z][A-Z]a' | ...
```

The first of these turns upper case letters into their lower case equivalents, and is obviously a useful thing to be able to do easily. The second is more of a joke: it replaces each letter by the next one in the alphabet to produce a very weakly obscured version of some text.

With the `-d` flag the command only needs to be given one string, and it just deletes any characters listed in this string from the file. For instance if you have received a file from a DOS/Windows site it probably has newlines represented as carriage return/linefeed pairs and it may be padded at the end with control-Z characters. It can be cleaned up using

```
tr -d '\015\032' < dos.file
```

where the bad news is that I have had to use octal escapes to specify the control characters that I want to discard.

A final use for `tr` is with the `-s` flag where if it finds any repeated strings of any of the characters from its second string it consolidates them into just a single instance. It can sometimes be useful to remove redundant blanks and newlines. Such processes are perhaps unexpectedly useful things to have in your armoury when you are building long pipes that use output generated by one utility as input to another: `tr` might be used to clean up the output to make it more digestible for the next use to be made of it.

```
... | tr -s '' ' \012' | ...
```

Not a very complicated tool, but useful to know about!

14 Data tools (2): `sort`

`sort` can read an input data set from standard input, or from a set of files specified on the command line. It then sorts all the lines in the data set according to various command line options, and sends the sorted data to standard output. `sort` uses an *external sort* algorithm, and is efficient even when dealing with large datasets — I use `sort` on datasets of several gigabytes quite regularly.

By default, the sort key is the entire line, but `sort` can be instructed to use a specific field in lines in the dataset using the `-k POS1[,POS2]` option. The default option is for fields to be delineated by whitespace characters, though this can be changed with the `-t seperator` option. `-k 2,4` would specify a sort key consisting of fields 2 through 4. Numeric (as opposed to dictionary) sorting order on a key can be specified with the `-n` option, and the sort order reversed with `-r`. Multiple sort keys can be specified on a single command line, with the primary sort key specified first. Sort can also be used to remove duplicate lines in the sorted output by using the `-u` option.

```
sort -t : -k2n -k1  <input
pear:3
banana:9
apple:70
banana:70
orange:100
banana:102
```

15 Data tools (3): `cut`

The `cut` utility can be used to extract sections from lines in its input. Like the other data manipulation commands it can take input from standard input, or from a list of files on the command line. `cut` can be operated in two basic modes: extraction of *fields* (`-f field_list`), and extraction of *characters* (`-c position_list`). The field list and character position lists are specified using a comma separated list of figures, with hyphens used to denote ranges. The command `cut -c 1-10,20-` would extract all but character columns 11-19 of the input data. When used for field extraction, the `-d delim_character` can be used to specify the field delimiter.

Eg, Using the same input as the previous `sort` example:

```
sort -t : -k2n -k1 input | cut -d : -f 2
3
9
70
70
100
102
```

16 Data tools (4): sed

The next data tool to mention is sed, which is one of the world's crudest and dullest editors. The reason is still exists and qualifies for inclusion in this course is that it is intended for use is embedded within scripts where it can do automatic and systematic editing for you. Thus nobody in their right mind would use sed when first typing in a program, but uses for it abound where a similar set of edits must be applied to a large number of files or where the nature of some wholesale change does not quite match the global-replacement facilities of your normal screen-based editor.

Since it is expected to be used in scripts, sed reads the original version of the file it is to process as a stream. It either reads from files listed on the command line, or if none are given it uses its standard input. The edited version is always sent to the standard output. Editing commands are either given directly on the command-line (typically delimited by quote marks) or in a script file that is named on the command line after them key -f.

When processing a file sed works through the file line by line applying each of its commands to all lines to which they are relevant. Thus really simple uses of sed let you perform collections of global exchanges all the way through a file. For instance to change every dog into a cat and every bone into a mouse you might use the command

```
... | sed 's/dog/cat/g; s/bone/mouse/g' | ...
```

The command being used is just "s" for substitute. The pattern that follows it is a regular expression (very much as for grep. After the final "/" there can be some flags, and the g here instructs that the exchange should be made as many times on the line as possible: by default only the first match on each input line would be processed.

Any sed command can be preceded by one or two addresses to limit the range within the file that it will be applied to. If you use a number as an address it is treated as a line number. The symbol $ stands for the end of the file. So if you want to make an edit that leaves the first dozen lines of your file alone you might try a command of the form 12,$s/xx/yy/. The more interesting feature of sed is that you can use regular expressions as addresses. If you write just one pattern it will apply its edit to all lines that match it. Appending an exclamation mark causes the edit to apply to all lines *except* those matching the pattern. A pair of comma-separated patterns will select a region from where the first pattern matches up to (and including) where the second does.

We have already seen the s command. Next I will mention that d deletes a line. It is then clear that the command

```
/public static void main/,/^ *}$/d
```

deletes all the lines from one containing the words about `main` down to one that consists of just a close curly bracket (possibly preceded by some spaces). Note that the range specified does *not* do anything to enforce matching of braces, so it will delete until the *first* close brace that it finds on a line of its own.

Another example, also slightly fragile, supposes that your code has a collection of lines that optionally print a trace statement, and that you want to comment them all out. Each such statement is supposed to be on just one line:

```
/if ?(debug)/{
s/^/  \/*--- /
s/$/  ---*\//
}
```

This applies a pair of commands to each line that contains either `if(debug)` or `if (debug)`[13]. The first command matches against the first position in the line and inserts there `/*---` while the second substitution puts `---*/` at the end of the line. You can see that a number of ugly backslash characters have been required so that various characters in patterns and replacement text are treated literally rather than as special. The use of multiple commands associated with one address expression requires that the braces appear on separate lines just as shown in this example.

Of course `sed` has many commands beyond the delete and substitute ones listed here, but its real power comes not from the richness of its command set but from the use of regular expressions. One way in which its variant on regular expressions differs from that used in `grep` is that you can enclose a section of a pattern within backslashed parentheses, and the effect is that whatever matches that pattern is captured and stored in a text variable. Up to nine such strings can be collected, and the replacement text in a substitute command can refer to them using `\1` to `\9`. So for instance if a file consisted of data in three columns separated by blanks it would be possible to rearrange columns ABC into the order BACA (making a second copy of the first column, A) using

```
s/ *\([^ ]*\) *\([^ ]*\) *\(.*\)/\2 \1 \3 \1/
```

This looks utterly horrible! But if read through one chunk at a time it can make sense! The pattern first looks for zero or more blanks. Then we have a block enclosed in `\(...\)` so that its value will be stored. The pattern within

[13]With a space in one but not in the other.

looks for an arbitrary repetition of characters that are *not* spaces. Next we match a further run of blanks, capture a second column of non-blank data, skip further inter-column space and at the end `\(.*\)` matches and saves everything up to the end of the line. That completes the pattern. The replacement text just plays back the saved material in the order we want it in. Achieving that effect using a typical mouse-driven editor would probably be pretty uncomfortable.

I think that typical strings of `sed` commands are dense enough in punctuation that they deserve to go into files so they can be prepared carefully and tested. Comment lines in such files begin with `#` and are somewhat desirable!

17 Data tools (5): `diff`

You probably already know that when you want to compare two files you use `diff`. By default it displays the differences between the two files by quoting line numbers and then listing lines that have been added, deleted or changed. A fully-fledged Unix tool-user will also be aware of at least some of the command-line options it can be given:

`-i` Ignore upper/lower case distinctions when making the comparison;

`-b` Treat multiple spaces as if they had been one space, and multiple newlines as if they had been just one. This can be useful if you have recently adjusted the layout in a file and do not want to be overwhelmed with reports that are just whitespace changes;

`-r` When the “files” given to `diff` are in fact directories this recurses through them and runs `diff` on all files in any common sub-directories;

`-D`*`symbol`* This merges the two files by inserting C-style conditional compilation directives of the form `#ifdef` *symbol*. It is arranged that if *symbol* is defined the generated file would be equivalent to the second file, while otherwise it would match the first one. Even if you are not using C this can be useful since you can edit the merged file and search for the string *symbol* to find where discrepancies had occurred;

`-e` Generate a simple edit script (intended for the Unix editor `ed`) that would change the first file into the second. In effect this is what is being done internally in `rcs` where is stores files as base versions and sequences of updates.

-u A GNU extension that produces *unified diffs*, a very useful form of diff output that is designed to be easy for humans to read and robust against slight version mismatches when used as input to the `patch` command.

Note that if the two files being compared have identical content then `diff` will generate no output at all. A handy tool built on top of `diff` is `rcsdiff`. This can be used to compare the current version of a file against any of the versions stored in the `rcs` repository. The default behaviour is to do the comparison against the most recently checked-in version, thus enabling a user to see whether a given file has been updated since.

18 `perl` as a super-set of `grep`

Thus far I have concentrated on tools that are fairly small and fairly specialised. Even when used individually these solve problems that would otherwise require a lot of manual work. Linked together with pipes they can do yet more. It is possible to use the Unix shell to run quite complicated sequences of sub-tasks, with conditional execution, recursion and most of the framework that you would expect in a real programming language. When used this way you would let the shell call upon the various lower level tools to perform each elementary operation: of itself it would do just the co-ordinating. Both the inefficiency of this and the fact that there are at least half a dozen different Unix shells in use has led to slightly more integrated scripting languages, such as `awk`, `perl` and `python` gaining popularity.

For processing a set of input data into a different form `awk` (or its slightly more powerful GNU cousin known as `gawk`) typically provides a compact yet powerful programming solution. For example, a program to generate a histogram from a set of experimental results could easily be crafted as just a few lines of `awk`.

When the task in hand requires interfacing with other system facilities (such as the OS, or other programs and libraries) then more powerful general purpose programming languages such as `perl` and `python` are best called upon.

The current lecturer of this course has a strong preference for `python` over `perl`, since it provides a modern object-oriented programming environment with support for exceptions and many other high-level features. `python` programs *tend* to be quite readable (and hence maintainable), but are often are a little longer than their `perl` equivalent.

In contrast, typical (badly-written!) `perl` scripts have a reputation for being “write-only” — their operation only understood at the time they were written, and by the person who wrote them. The is perhaps because `perl` makes widespread use of significant punctuation characters, has a huge array of predefined cryptic operators and functions with strange default behaviours — all designed to save the programmer precious key strokes. However, despite my views, `perl` has been around a long time, and is widely used and installed on many systems — more so than `python`, currently. Hence its (slightly reluctant) inclusion in this course...

`perl` can simulate and generalise most of the tools mentioned so far, including quite elaborate shell scripting. Since it is a full-scale programming language I will not start to pretend to cover more than a very tiny fragment of it here. I can suggest the introductory book by Schwartz and Christiansen[3] and leave the definitive guide by Larry Wall[4] for those who want to become real experts. I will illustrate the language by giving `perl` scripts that simulate behaviour similar to that which can be obtained using one of the more specialist tools. So first I will cover `grep`.

The first version of this shows the "program" passed to `perl` on its command-line after the key `-e`:

```
perl -n -e 'print if /regexp/' < somefile.txt
```

In this very concise example the `-n` flag causes the command given following `-e` to be applied to every line of the input. The command given prints a line if it contains a match against the given expression.

The same effect can be expanded out into something that looks more like a program: A `perl` script lives in a file, and contains

```
#!/usr/bin/perl -w
while (<>) {
   if (/regexp/) {
      print $_;
   }
}
```

When the name of this file is presented to the shell it reads data from either each file named on its command line, or if there are none of those from its standard input. The first line marks it as a `perl` script[14], and the `-w` flag asks for warnings about dubious constructs. Since almost anything is valid in `perl` you have to do something quite seriously weird to get much of a warning! The body of the program we have here is a loop. The angle operator (`<>`) causes the next line of input to be read and saved in a standard place. The `if` statement matches this text against the given regular expression (and

[14]Well, you need to indicate there whatever place on your Unix system `perl` is actually installed.

regular expressions in `perl` are as extended as you could ever dream of). If there is a match then the print statement is activated to send something to the standard output. The "`$_`" references the location where the most recent input line had been stored. The effect is that all lines matching the regular expression get printed.

It is perhaps obvious that there will then be arithmetic statements, variables, arrays and the ability to nest arbitrary mixtures of conditionals and loops. You can define `perl` procedures and call them recursively. In consequence and at the cost of writing a slightly longer script `perl` can perform much more complicated pattern detection than can `grep`. But because the full power of regular expression matching is available within `perl` those tasks that just need that remain quite easy to express.

19 `perl` for scripting

There are three things that `perl` does that make it a really useful language for writing system maintenance utilities in. The first is that it has a rich built-in collection of directory and file manipulation primitives. If the variable `$filename` is the name of a file you are interested in then there are over two dozen tests you can perform on it. Note that the operators are the same as for the previously mentioned shell command, `test`. Important examples are:

`-r $filename` yields true if the file exists and is readable;

`-w $filename` is true if the file is writeable;

`-e $filename` is true if it exists;

`-z $filename` is true if it exists but has zero size;

`-d $filename` if it is a directory rather than a simple file.

and other similar short operators can find the age of a file or the size (in bytes) that it currently has. There are then further facilities for traversing directories, expanding file-name wild-cards and changing file-access permissions.

The second feature is that it is easy to launch sub-processes and retrieve their output. If you want the output from the program you are going to run to be sent to a file or to the standard output you can use a `system` function, as in

```
system "javac Testfile.java";
```

but on other occasions you want your `perl` program to capture, parse and respond to the output generated by the program. In such cases you just write an expression which is the command you want executed enclosed in backquotes. In my example here I just print the output straight away — normally you would store it in a variable and process it further:

```
print `javac Testfile.java`;
```

The final strength that `perl` brings is the use of regular expressions to decompose program output and a neat (?) syntax that lets you separate out the parts of your sub-program's output. In fact this just comes from its general pattern-matching ability, but I will illustrate it here based on a sub-call to `grep`. Actually one would normally do `grep`-like things *within* your `perl` program, but I just want an illustration of calling a program and then parsing the output. So suppose I have a file called `"funny.data"` and within it there are a number of lines that contain the string `XXX`. I can cause `perl` to invoke `grep` by putting

```
`grep -n XXX funny.data`
```

in my perl script. With the `-n` flag the output from `grep` might be something like

```
108:    while ((c =  XXX getc(fmakebase)))
150:        return 1; XXX
179:      fXXXor (i=0; i<n_user_words; i++)
```

with a line number, then a colon, and then the line that the pattern was found in. The output from `grep` gets passed back to `perl` as an array of lines, and within each line it is useful to search (using a very simple regular expression) for the colon and split the line into two parts there. As a simple if frivolous demonstration I will just filter things so that I only display things that are found within the first 160 lines, and I will annotate the output a little:

```
#!/usr/bin/perl -w
@grepresults = `grep -n XXX funny.data`;
chomp @grepresults
foreach $line (@grepresults) {
  ($linenumber, $contents) = split(/:/, $line);
  if ($linenumber < 160) {
     print "line=$linenumber, data=<$contents>\n";
  }
}
```

The output might then be

```
  line=108, data=<    while ((c =  XXX getc(fmakebase)))>
  line=150, data=<        return 1; XXX>
```

A more realistic application that might start off in a similar style would run a compiler and then retrieve and parse the error messages for you…

20 `perl` as a general-purpose language

Again I must stress that this course is at best a taster for `perl`, and so rather than giving a proper presentation of its syntax and capabilities I will just give a couple of annotated sample programs. The ones I present are taken from the suggested book[3].

20.1 Counting repetitions

My first example program counts the number of times each word in present in a file, where the file contains just one word per line:

```
  #!/usr/bin/perl -w
  chomp(@words = <STDIN>);
  foreach $word (@words) {
        $count{$word}++;
# or:   $count{$word} = $count{$word} + 1;
# or:   $count{$word} += 1;    (all 3 are equivalent)
  }
  foreach $word (keys %count) {
     print "$word was seen $count{$word} times\n";
  }
```

The first line is one we have seen before and marks this as a `perl` script. On the next line `@words` refers to a variable that is an *array* type, and when we assign to it from `<STDIN>` it ends up with each line of the input file in a separate element. The built-in function `chomp` removes the newline characters that may initially be present in this array. The `foreach` statement iterates over all items in the array. The reference `$count{$word}` is the use of a *hash table* which is a bit like an array but can be indexed by arbitrary things (in this case our words) not just by numbers. Note that in `perl` references to scalar variables include a `$` in the variable name to show that a scalar rather than an array value is involved. Note also that the table of counts did not need to be initialised first: in an arithmetic context an unset value is treated[15] as zero! Finally the built-in function `keys` is used on the hash table of counts. In this case it is necessary to reference the whole hash table, not just some entry in it, and this is why the text reads `keys %count` with a percent mark. Observe that the output text is created by having variable references expanded within the string.

20.2 Extracting data from a table

Suppose you have a file whose contents are arranged as a number of fields separated by colons. The fifth such column contains a name, which is followed (optionally) by an address and phone numbers (separated off by commas). This curious format is illustrated by a two-line (non-real) file:

```
acn1:x:1000:1000:Arthur C Norman,T34,,:/home/acn1:/bin/bash
am:x:1003:1003:Alan Mycroft,,,:/home/am:/bin/bash
```

and happens to follow the layout used by Unix for password files. The task is to list just the first names of all the users present. This provides an illustration of the `split` operation. This takes a regular expression and a string and creates an array whose elements are the parts of the string delimited by things that match the regular expression:

```
#!/usr/bin/perl
while (<STDIN>) {
  chomp;
  ($gcos) = (split /:/)[4];
  ($real) = split(/,/, $gcos);
  ($first) = split(/\s+/, $real);
  print "$first\n";
}
```

This is tolerably dense, but I hope that with some explanation it will become readable. It starts by reading in lines from its standard input. The `while` loop reads one line at a time. The line that has been read is left in a default variable called `$_` and other operations work on this if not told to use something else. This saves typing somewhat. As before `chomp` removes the newline that is at the end of each line. Here it works on `$_`. The right hand side of the next line starts by splitting the input line at each colon. This hands back an array and so the element with index 4 is selected. Since

[15] However here if we use the `-w` flag we will be presented with a bunch of warnings when this happens. I rather suspect that a normal `perl` style does not do the initialisation and only uses the `-w` flag while testing its code. However a line `if (! defined $count{$word}) {$count{$word} = 0;}` could clean things up.

the first item has index 0 this gets the field that we want. It is assigned to `$gcos` where this name hints at a traditional Unix name for this field in a password file. The next use of `split` find a comma, and can omit an explicit subscript `[0]` because some cleverness in the assignment puts the first item from the right-hand-side array into the scalar variable `$real`. The final use of `split` uses a regular expression that looks for a string of one or more non-space characters. The escape sequence `\s` is one of very many pre-defined escape sequences that give you very concise ways of specifying words, numbers, whitespace and other common things you may need to match. Finally I just print the result.

20.3 A bit more pattern matching

In a bit of `perl` pattern matching you can enclose parts of your regular expression in parentheses, and the result (without backslashes this time) is that the parts of the pattern that match those little fragments get stored in variables for you. For instance when you use the C shell the display you get when you use `time` to record how long a command takes to execute looks something like:

```
44.2u 1.8s 0:47.85 96.3% 0+0k 0+0io 24099pf+0w
```

where the fields show user-mode time, operating system overheads, total elapsed time and various statistics relating to memory and processor utilisation. I present this example even though it comes from a different shell because it provides a plausible example of some murky text string that needs to be decomposed. To skip the issue of capturing the data I will put it into a `perl` variable by hand. In this display I will underline the parts that I will suppose are wanted (for some reason):

```
$data = "44.2u 1.8s 0:47.85 96.3% 0+0k 0+0io 24099pf+0w";
                      ==    ====             =====
```

The relevant fields can be extracted by matching the data against a regular expression

```
$data =~ /:([^.]*)\S*\s([^%]*)%.*io (.*)pf/;
print "field1 = $1  field2 = $2  field3 = $3\n";
```

The operator `=~` asks for a pattern match. Within the horrid looking regular expression there are a couple of patterns that represent ranges of characters, so for instance the sub-pattern `[^.]*` is looking for an arbitrary number of characters that are not dots. The predefined escape `\S` matches any character that is not whitespace. And then some parts of the regular expression are in parentheses so that the corresponding fragments of matched text can be retrieved as `$1` etc in what follows. Perhaps a little cryptic but very powerful!

There are many hundreds of `perl` modules available on archive sites for downloading. These provide ready-written facilities for a huge range of operations you might imagine wanting: image compression, network management, database connectivity, extra data-types and algorithms... the list goes on and on. One of the strengths of the language is the wide range of things that you do not have to write for yourself but can pick up in this way.

A context where you are especially liable to hear `perl` mentiones is that of so-called `cgi-bin` scripts, which are ways in which a web-server can run programs in response to clicks made by the remote user who is browsing a page. `perl` sees major use as a programming language in this special but very important niche.

21 A brief note about `emacs`

There is one tool that is very widely used on Unix (and of course elsewhere) which draws on some of the tradition of these tools, but which embeds all of its power within a single comprehensive interface. This is `emacs`. If you are only using this as a simple screen editor, clicking on your mouse every time you want to open or save a file and doing most of your navigation by dragging a scroll-bar then you are not following the use-pattern that the original `emacs` designers had in mind and you may be missing out on understanding just how much it can do for you.

`emacs` provides an astonishingly large collection of editing commands, various of them based on searches using regular expressions of the form we have seen here. It can also provide a model for file-management based on the idea that you are "editing" your directory structure, and it provides an editor-like environment for reading e-mail (which thus naturally permits easy searches through incoming or stored mail, and makes the commands to display or delete mail rather like those to display or delete files). Its scripting facilities are based on it having an embedded language — `emacs lisp` — which makes it possible to write quite general programs based upon the basic (and indeed not so basic) editing commands that are pre-defined. Perhaps the best illustration of the power and benefit of the scripting capabilities is the way in which the editor can be customised for the language that you are editing, so that there are simple commands to move across blocks and procedures, to recompile parts of your code and to re-position the editing focus at the first place that the compiler spotted a syntax editor. Simple users can of course benefit from these language-specific `emacs` modes without understanding where they came from, but the open nature of emacs means that those who are more experienced can first customise existing modes, and then design new ones to suit their own exact preferences or to perform new tasks.

My belief is that gaining initial experience with `grep` and the like is useful even if you intend to end up as an ultimate `emacs` wizard who uses it for everything.

On some occasions, you may find that you want to perform some minor edit on a file that doesn't really warrant the overhead of "firing-up" `emacs`. The editor `vi` can prove useful in such instances. The document can be navigated using the cursor keys, or if the terminal is not setup correctly, 'h','j','k' and 'l' can be used instead. Ctrl-U and ctrl-D move up and down the page. To insert text at the cursor position hit 'i' then enter the text. Insertion mode is exited by hitting ESC. To append after the cursor position (eg to add something to the end of the line) use 'a' instead of 'i'. Characters can be deleted with 'x', whole lines with 'D'. Searching can be achieved using '/searchstring' to search forward, '?searchstring' to search backwards. The next match is found by hitting 'n'. After editing a document, a save and exit can be performed using ':wq'. Editing can be aborted with ':q!'. `vi` provides many quite powerful features, but for those you'll have to refer to the man page.

22 "Unix" tools and Microsoft Windows

The Unix tools discussed here represent a particular perspective on the world. It starts with an expectation that serious users are going to be willing to take time to learn how to use the systems that are at their disposal. It is perhaps rooted in the "no pain, no gain" world view, which expects that anything that goes truly out of its way to be easy for an utter novice to use will necessarily only have limited capabilities. An elaboration on this is that the Unix tools are attempting to provide fundamental building blocks of functionality (such as pattern matching), rather than complete solutions (like writing your program for you automatically). Big tools will be wonderful when what you want to is something that has been thought of and supported by the tool author, but when it does not you are in trouble. Small but generic tools provide at least some leverage even in new or unusual situations.

A second part of the Unix philosophy is that it should be possible to automate tasks. Again setting up the scripts with all their pipes, redirections and messy parameters may be a painful cost, but is can be seen as an investment since then the operations concerned can be performed over and over again with the greatest of ease. This applies just as much to slightly complicated and repetitive edits on a large file as it does to the process of recompiling your program, running all your standard tests and checking their output to see that it is as expected. For tiny tasks where you write a short program, run it once and do not worry about documentation all this seems unnecessary, but for bigger projects it is of course vital.

A third aspect of Unix is perhaps a legacy of the fact that most of these tools have been around for a long while. However the tools described here represent an emphasis on command-line working, with hands firmly positioned above a keyboard. For one who has taken the time to learn how to use the tools and who has set up a set of well designed scripts and `Makefile`s almost all common operations can be performed using a quite short sequence of keystrokes. Hands do not have to move between keyboard and mouse, and the accurate positioning needed when selecting test visually is avoided. The shell history mechanism saves keystrokes compared to the DOS/Windows equivalent that tends to lead to long sequences of repeated presses on arrow keys.

A final thing to note about these Unix tools (and to contrast, if you have used it, with the Microsoft Development Studio) is that the tools I have discussed here are by and large neutral. `make` does not mind whose compiler it is going to invoke. Indeed it can be used to automate anything where one file gets re-built from another and where date-stamps define dependencies. `rcs` stores updates to your files, but leaves you to decide which editor you will use to change them. It does not lock you in to one vendor or style. `grep`, `sed` and the rest are all general purpose.

Some tasks, however, do not fit in at all well with this style of work. The prime example is probably desk-top publishing where fine adjustment of the visual appearance of documents matters, and the automated testing of windowed applications where text-based scripts find it a little hard to simulate full interactive input and to check that the programs's behaviour is correct.

23 Conclusion

As previously mentioned, all the "Unix" tools are in fact pretty generic, and although they original arose as part of the Unix project they have been ported to other platforms, notably Windows. There (of course) they are run from a command-line. Amazingly you can even get a version of the `bash` shell that runs under Windows, although I would suggest that if you really want that environment you should probably install Linux on your computer instead! Better than the fact that they are available is the fact that there are generally *free* versions of everything, in particular versions from the Free Software Foundation issued under the GNU public license. You should perhaps be aware that there are also a load of slightly half-baked approximations to the real Unix tools out there too, where somebody has wanted (say) `grep` and implemented some subset of it for themselves. You may like to try to ensure that you find definitive versions of at least the more important tools. A useful and fairly complete collection of tools is CygWin:

```
ftp://sunsite.org.uk/Mirrors/sourceware.cygnus.com/pub/cygwin/
```

Cygnus spearhead a project intended to make it easy to host programs that were originally written for Unix on a Windows machine, and in the process they have gone quite a long way towards making it possible to simulate a Unix development environment within Windows. They have an implementation of `bash` as well as pretty well all the tools discussed in this course, and the associated free C, C++ and Fortran compilers are of respectable quality.

Everything you might want relating to `perl` is at

```
http://sunsite.doc.ic.ac.uk/packages/CPAN/
```

where perhaps the sub-directory `ports/win32/ActiveState` may be of interest. Note that as with all references to material on the web there is little guarantee of stability in what you will find, but I would expect that the mirrors at SunSITE would normally hold copies of all the most commonly useful and most robust software collections.

24 Bibliography

References

[1] Tom Christiansen. Csh programming considered harmful. http://www.mars.org/home/rob/docs/csh-whynot.html.

[2] Daniel Gilly. *Unix in a Nutshell.* O'Reilly & Associates, 1992.

[3] Randal L Schwartz and Tom Christiansen. *Learning Perl.* O'Reilly & Associates, 1997.

[4] Larry Wall, Tom Christiansen, and Randal L Schwartz. *Programming Perl.* O'Reilly & Associates, 1996.