Introductions to Computing

MJ Rutter
mjr19@cam

Lent 2009

Scripts

Laziness

Laziness is a great asset, and using scripts to automate repetitive tasks is extremely sensible. If well written, they should be less error-prone than humans too.

Whereas Fortran is ideal for CPU-intensive numerics, scripts are excellent for file and string processing on a lighter scale. Even, as shall be considered later, a moderately heavy scale.

First, What You Already Know

The most attractive language for scripts is `bash`, because you already know much of it. A `bash` script is very little different from the commands one types interactively.

Note that whilst `csh` was primarily designed for interactive use, the Bourne shell `sh` was designed primarily for scripting. In the Bad Old Days one needed to know both, as using `sh` interactively would cause madness. The advent of `bash`, a superset of `sh` which is perfectly usable interactively, destroyed any need for the various `csh`-derived shells (now mostly `tcsh`). Both the Bourne shell and `csh` appeared in about 1978. Ten years passed before `bash` appeared, and several more before it became widespread.

Before 1978 UNIX suffered the Thomson shell, which was also called `sh`. It was dreadful.

The first mainstream shell to support filename completion and command-line editing with the cursor keys was `tcsh`, an enhancement of `csh`, which appeared in stages in the early to mid 1980s. So for a half a dozen years any UNIX user wishing to remain sane used `tcsh` interactively, and the Bourne for scripts, until `bash` brought command-line editing to the Bourne shell.

The Very Basics

It is important to understand how the shell handles command line arguments before one gets very far. It is best to write a simple program to experiment with. There are fewer traps if one writes it in C,

```
#include<stdio.h>

int main(int argc, char** argv){
  int i;

  for(i=0;i<argc;i++)
    printf("Argument %d is *%s*\n",i,argv[i]);

  return 0;
}
```

C Haters Club

Those who would prefer a nearly-equivalent shell script might like

```
#!/bin/bash

for arg in "$0" "${@}"
do
  echo "Argument *${arg}*"
done
```

Spaces

```
$ ./a.out Hello World
Argument 0 is *./a.out*
Argument 1 is *Hello*
Argument 2 is *World*
$ ./a.out Hello     World
Argument 0 is *./a.out*
Argument 1 is *Hello*
Argument 2 is *World*
$ ./a.out "Hello World"
Argument 0 is *./a.out*
Argument 1 is *Hello World*
$ ./a.out "Hello     World"
Argument 0 is *./a.out*
Argument 1 is *Hello     World*
```

More Spaces

```
$ ./a.out 'Hello World'
$ ./a.out Hello\ World
$ ./a.out Hello" World"
$ ./a.out Hello' 'World
$ ./a.out Hell"o W"orld
```

All of these are equivalent, and set argument 1 to 'Hello World'. The program or script run has no idea which version was used.

Preceding a character by a backslash in general supresses any special meaning it has, and includes it literally. Here the special meaning of argument separator is being supressed.

A string with double quotes supresses the special meaning of spaces, whereas single quotes supress rather more (see later).

Spaces Alone

```
$ ./a.out Hello " " World
Argument 0 is *./a.out*
Argument 1 is *Hello*
Argument 2 is * *
Argument 3 is *World*
$ ./a.out Hello "" World
Argument 0 is *./a.out*
Argument 1 is *Hello*
Argument 2 is **
Argument 3 is *World*
```

An argument can be a space, or even nothing.

Variables

The variables `$0` to `$9` give the first nine arguments. For the rest, use `shift`, which leaves `$0` (the script name), but discards `$1` and moves everything else down one.

Setting other variables is simple: `name=value`
However, 'value' must be a single word, not
`x=Hello World`
but
`x='Hello World'`

Environment variables are also inheritted by a process's children, whereas simple shell variables are not. To turn a shell variable into an environment variable, `export` it.

```
$ foo=bar; sh -c 'echo $foo'

$ foo=bar; export foo; sh -c 'echo $foo'
bar
$ foo=bar sh -c 'echo $foo'
bar
```

`sh -c` – execute command in new shell
`var=value cmd` – set the environment variable `var` for the execution of `cmd` only, and then return `var` to its previous state.

Variables, Spaces, and Arguments

```
$ x="Hello World"
$ ./a.out $x
Argument 1 is *Hello*
Argument 2 is *World*
$ ./a.out "$x"
Argument 1 is *Hello World*
$ ./a.out '$x'
Argument 1 is *$x*
```

The Bourne shell's use of quotes is fairly consistent. Within single quotes nothing is special. Within double quotes, `$`, `` ` `` (see later under 'third quote') and `\` are special. Users of `csh` find a mess: the rules are the same at a first glance, but

```
> echo nuts$
nuts$
> echo 'nuts$'
nuts$
> echo "nuts$"
Illegal variable name.
```

Shell Wildcards

The space is not the only character which the shell treats specially. So too `$` (as above), and the wildcards `*` and `?` along with square brackets.

These characters are odd. They are nothing to do with the regular expressions used by awk, emacs, grep, perl, python, sed and vi. Unfortunately one needs to know both.

They usually refer to filenames, and `?` stands for any single character except / and except a leading dot. The `*` stands for any number of any characters, with the same two exceptions.

However, if there is no file name match, they do nothing special.

Really Wild

```
$ echo ok?
ok?
$ touch oki
$ echo ok?
oki
$ touch oka
$ echo ok?
oka oki
$ x='ok?'
$ echo $x
oka oki
$ echo "ok?"
ok?
```

The use of `'ok?'`, `ok\?` or `ok'?'` would all be equivalent to the last line.

The `touch` command will create an empty file of the given name.

The `oka oki` example produces two separate arguments, not one.

Stupid Names

```
$ ls *
ls: illegal option -- -
ls: illegal option -- v
Usage: ls [-aAbcCdDfFgilLmnopqrRstux1] [file ... | directory ...]
$ ls
--invalid  correct
$ rm --invalid
rm: illegal option -- -
usage: rm [-efirR] file ...
```

File names starting with a hyphen are a pain. They are likely to be considered options by most commands. Afterall, wildcards are expanded by the shell before the command is called. Using quotes will get nowhere (they are useful for filenames containing a *).

Two tricks. Make sure that the name starting with the hyphen appears after an argument without a hyphen.

```
$ touch x
$ rm x --invalid
```

Or hope that some commands accept two hyphens as meaning end of options.

```
$ mv -- --invalid sensible
```

Multiple Stars

Whereas wildcards never match leading dots, nor forward slashes (directory separators), they can be used multiple times. So

```
$ echo */*.tex
```

will show all files ending `.tex` at one directory level below the current one. Similarly

```
$ wc -l */*.tex
```

will give their lengths in lines, and

```
$ grep Thatcher */*.tex
```

will find all references by surname to St Margaret of Finchley.

The Third Quote

Not a bizarre purchasing requirement, but an acknowledgement that the standard 7-bit ASCII character set contains three different quotes: double quote, closing single quote (or apostrophe) and opening single quote. LaTeX expects quotes to be used in contrasting pairs (as does English), but the shell expects them in matched pairs.

The meaning of the back quote (or opening quote), as found at the top (or sometimes bottom) left of the keyboard, is execute the command enclosed, capture `stdout`, and replace the region within the quotes by `stdout`.

Too Much Expansion

The use of * or backquotes can give very long arguments.

Bourne Shell

```
$ /bin/echo `perl -e 'print "A"x20000;'` | wc -c
     20001
$ /bin/echo `perl -e 'print "A"x40000;'` | wc -c
/bin/echo: arg list too long
         0
$ /bin/echo `perl -e 'print "A "x20000;'` | wc -c
/bin/echo: arg list too long
         0
```

csh

```
% /bin/echo `perl -e 'print "A"x20000;'` | wc -c
Word too long.
         0
% /bin/echo `perl -e 'print "A "x8000;'` | wc -c
Too many words from ``.
         0
```

Keep Directories Small!

These errors are coming from the shell, not `echo`, so will apply whatever command was used.

The precise point at which things fail will vary, but no shell I have tested will cope with 17,000 arguments. The most accommodating copes with up to 16,382 arguments, and a total argument length of about 130,000 characters. The least accommodating failed to cope with 6,900 arguments, or a total length of 40,000 characters.

It is thus easy (though not sensible) to create a directory in which '`rm *`' will simply fail because the star will expand to something over this limit.

(The traditional UNIX filesystem (including ext3) uses unindexed flat files for directories. These become rather inefficient with more than about a thousand files in them anyway.)

For Loops

```
for party in Labour 'Liberal Democrat' Green
do
  echo "Don't vote $party"
done
```

The syntax of the `for` statement is
for *variable* in *word-list*

The more usual use would be something like
`for file in *.dat`

`for` and `seq`

Some UNIXes have the command `seq`, which simply produces a sequence of numbers.

```
$ seq 3 5
3
4
5
```

This would be useful with the `for command`:

```
for n in `seq 3 2 13`
do
  echo "Engineers believe $n is prime"
done
```

However, too many UNIXes don't supply `seq` for it to be useful. MacOS is one which does not.

`seq [start [inc]] end` with `start` and `inc` defaulting to one if not given.

Conditions

The Bourne shell has a standard if construction:

```
if expr
then
  ...
elif expr
then
  ...
else
  ...
fi
```

There can be any number of `elif` clauses (including zero), and the `else` is optional too.

Truth

Almost every programming language regards zero as false, and not zero as true. All shells do the opposite – zero is true, and non-zero false.

The logic of this is that if a process's exit code is zero, this indicates success, whereas non-zero indicates failure, with the precise value perhaps giving further indication of what the problem was.

```
$ grep root /etc/passwd
root:x:0:0:root at pc0:/root:/bin/bash
$ echo $?
0
$ grep secret /etc/passwd
$ echo $?
1
$ grep secret /womble
grep: /womble: No such file or directory
$ echo $?
2
```

More `if`s

Although `if` is often seen in conjunction with `test`

```
if test "$1" = "-help"
then
  echo "Ha! Ha!"; exit
fi
```

and then most often with `[` used as a synonym for `test`

```
if [ "$1" = "-help" ]
```

it can be used with any command.

If `[` is used in place of `test`, then the expression must end with a `]`, and there must be whitespace on either side of the brackets. The command `/usr/bin/[` does really exist, although it will also be built in to any sane shell.

Other Common Tests

`a = b`	string equality
`a != b`	string inequality
`-n a`	string is of non-zero length
`-f name`	file exists as regular file
`-r name`	we have read permission to file
`-w name`	we have write permission to file
`-s name`	file is of non-zero length
`-d name`	directory exists

Also the integer numeric comparisons -eq, -ne, -gt, -ge, -lt, -le.

Modern UNIXes, and `bash`, add `a -nt b` file a is newer than file b

Tests can be combined with `-a` (and) and `-o` (or), and negated with `!`.

```
if [ -s $input -a -r $input ]
```

(We have a readable input file of non-zero length.)

Silly Tests

```
if [ $1 = -help ]
```

If no first argument exists, this expands to

```
if [ = -help ]
```

which is not false, but rather a syntax error.

```
if [ $1 -eq -help ]
```

When interpreted as integers, '-help' is zero, and `$1` is very likely to be too.

In the first example, putting the `$1` in double quotes means that it expands to the null string if unset, which is fine. The other work-around sometimes seen is the bizarre

```
if [ x$1 = x-help ]
```

Other ideas for `if`

```
if grep -q CASTEP $file
then
  :
else
  echo "$file is not a Castep output file"
fi
```

`grep -q` writes nothing to `stdout`, but returns the usual exit codes.

If using `bash`, the more compact

```
if ! grep -q CASTEP file
then
  echo "$file is not a Castep output file"
fi
```

is possible.

Other `case`s

```
case "$type" in
  -gif)
    convert="ppmtogif"
    ;;
  -jpeg)
    convert="cjpeg -optimise"
    ;;
  -jpeg=*)
    q=`echo $type | sed 's/-jpeg=//'`
    convert="cjpeg -optimise -quality ${q}"
    ;;
esac
```

This sets `$convert` to be a suitable command for converting a ppm stream to the desired format.

Note that each `case` clause ends with `;;`, and the usual shell wildcards are available, with the added quirks that forward slashes and leading dots are matched just like any other character, and `|` separates alternatives, e.g. `-h|--help)`.

Short ifs

The logical operators `&&` (and) and `||` (or) can be used to join two commands. As they short-circuit, they are usually used to produce rather terce if statements.

```
[ -w $output ] || exit 2
```

```
[ "$1" = "-v" ] && set -x
```

If a conditional expression short-circuits, then it is evaluated left to right and exits as soon as the result is known. For instance, in the expression ‘foo or bar’, if foo is true, the expression must be true, and bar need not be evaluated. The shell, and C-like languages, guarantee to do this. Fortran does not.

The advantage of not short-circuiting is that the language can choose to reverse the order of evaluation if one half of the expression is simpler to evaluate than the other, or it can evaluate both at once (perhaps to reduce CPU pipeline stalls due to data dependencies within the individual expressions).

So writing things like

```
i>0 && j/i>20
```

is safe in C as `i` cannot be zero when `j/i` is evaluated. In Fortran it is not safe.

More Loops

There is also a while loop. It is often used in conjunction with `read`, which reads a line from `stdin`, returning an error on end of file.

```
while read line
do
  echo Processing $line
done
```

File Conversion

```
for bmp in *.png
do
  targ=`echo "$bmp" | sed 's/png$/eps/'`
  bmp2eps $bmp > $targ
done
```

If using `bash`, then

```
  targ=${bmp%png}eps
```

is shorter, though the overhead of `sed` is insignificant compared to that of `bmp2eps`. A middle way would be

```
  targ=`basename "$bmp" eps`png
```

Note that the above can be entered interactively, pressing {return} after each line, and even recalled and edited by pressing cursor up. This is not possible in `tcsh` with its corresponding `foreach` construction.

Which of the above are safe if a filename contains a space?

Finding PostScript Bounding Boxes

In PostScript, a comment line gives the size of the PostScript image, in points. As a PostScript may include lots of other EPS files, each of which may still have its own bounding box comment present, the first line starting
`%%BoundingBox:`
is the one which covers the whole document. Unless that line is
`%%BoundingBox: (atend)`
in which case the last such line is the relevant one.

The line contains four integers, giving the lower left and upper right corners of the image:
`%%BoundingBox: llx lly urx ury`

I should add at this point for the unbelievers that PostScript is merely another text-based computer language like C, Fortran or sh. It is usually machine-generated (unlike the above), but it can be written from scratch, and the machine-generated sort can be editted.

Finding the Bounding Box, version 1

```
bb=`grep '^%%BoundingBox' $infile | head -1`

if [ "x$bb" = "x" ]
then
  echo No BoundingBox found in $infile 1>&2
  exit
fi

set $bb

if [ "$2" = "(atend)" ]
then
  bb=`grep '^%%BoundingBox' $infile | tail -1`
  set $bb
fi
shift
```

The `set` command will set `$1` to its first argument, `$2` to its second argument, etc., discarding the original arguments to the script.

And Now for Something Completely Different

```
continue=:
bb=
exec < $infile
while $continue
do
  read x y || continue=false
  case $x in
  %%BoundingBox:)
    bb=$y
    continue=false
    ;;
  esac
done
set $bb
```

This should give similar results to

```
bb=`grep '^%%BoundingBox:' $infile | head -1`
set $bb
shift
```

The `exec < $infile` redirects `$infile` to `stdin` from that point onwards.

Note for pedants: dropping the `exec < $infile` in favour of adding `< $infile` to the `done` will work in `bash` and the POSIX version of the Bourne shell (including `ash`), but will fail in the original Bourne shell (as used on Tru64 and Solaris 9 at least).

Maths

The maths abilities of the Bourne shell are approximately zero. Even addition requires one to use `bc`, or, for integers, `expr`. So one sees:

```
x=1
while [ $x -lt 50 ]
do
  echo "Geometric series are fun. $x"
  x=`expr $x '*' 2`
done
```

This is awkward, not least because `expr` needs careful thought.
`expr "$x * 2"` is not correct, as three distinct arguments are required here.

Bourne Again

The `bash` shell's most useful extensions over the original Bourne shell are in the areas of arithmetic and regular expressions.

Integer arithmetic expressions can be enclosed by `$((` and `))`, so the `expr` command above becomes `x=$(( $x * 2 ))`
(and the asterix does not need quoting within the double brackets.

Regular expressions are available through an enhanced test.

```
expr "$1" : '[0-9]*$' > /dev/null && echo +ve int
```

can now be written

```
[[ $1 =~ ^[0-9]*$ ]] && echo +ve int
```

The match operator, `:`, of `expr` has an implicit leading `^`. As `expr` reports to `stdout` the number of characters matched, this output is thrown away to `/dev/null`.

Note that within the `[[ ]]` construction quotes are not necessary. Indeed, placing single quotes around the regular expression causes it to fail.

Internal vs External

```
$ cat add.sh
#!/bin/bash
x=0
while [ $x -lt 1000 ]
do
  x=`expr $x + 1`
done
echo $x
$ /usr/bin/time ./add.sh
1000
  2.19 real   0.61 user    1.48 sys
```

Replacing the `expr` command with the equivalent `$(( ))` command, and increasing the loop count to 100000, changes the time to

```
  3.53 real   3.20 user   0.31 sys
```

Even the simplest possible external command, `/usr/bin/true` called with an explicit path, takes 1.4ms to execute on this 2.4GHz computer. Fine if called a few times, but awkward in a loop with a high iteration count.

Worse External Problems

On many computers, when a command exits a record is written to the process accounting file. This record is typically around 64 bytes, so the wrong sort of loop above causes this file to grow at several KB per second. Which is inconvenient if anyone wishes to use it for debugging, for a shell script calling lots of external commands in a loop can easily dump tens of thousands of records into it.

What is worse is that the program may well barely show in the output of `top`, for much of the time is spent in the thousands of short-lived external commands, not the script's own process. The people who write such things are often the same people who believe that `top` lists all relevant running processes, so they don't spot their errors...

Note the difference between `ps aux`, which shows the CPU time used by processes, and `ps auSx`, which shows the CPU time taken by processes and any of their children which have exited. These can be very different for shell scripts.

The Other Problem: Nasty Internals

```
$ man kill
...
       kill -t [SIGNAL]...
...
$ kill -t
bash: kill: t: invalid signal specification
$ /bin/kill -t
 1 HUP       Hangup
 2 INT       Interrupt
 3 QUIT      Quit
...
```

The man page describes the command in `/bin`, but `bash` (and `tcsh`) have a built-in `kill` command with a different syntax.

The same problem applies to `echo` and `time` and, for `tcsh` users, `nice` and `renice`. In the Bourne shell, `test` and `[` are builtins.

If `kill` were not a shell built-in, then, once one had reached the limit of the number of processes a user was permitted to run simultaneously, it would be impossible to use `kill` to terminate any of them.

Shell Functions

```
yesno(){
  echo "$1" '(y/n)'
  read ans

  case "$ans" in
  y|Y)
    return 0
    ;;
  n|N)
    return 1
    ;;
  *)
    echo 'Please answer y or n'
    yesno "$1"
    ;;
  esac
}

yesno "This lecture is too long" && \
  echo "Go and read the Tony Benn diaries"

if yesno "Margaret Thatcher was right" ; then
  echo "No, She was entirely centralist"
else
  echo "But She was, is, and shall be, correct"
fi
```

More Functions

Function arguments are passed as `$1`, `$2`, etc.

Variables are global between the main shell program and its functions, with the exception of `$1`, `$2`, etc., which are restored to their original values when the function returns.

The return value of the function is the return value of the last command (or function) executed in it, unless the exit is via an explict `return`.

It is possible to declare variables as local to a function body in `bash`.

Functions can be used as signal handlers (and `bash` allows signal names, as well as numbers, here).

```
#!/bin/sh
ouch() { echo ouch; }
trap ouch 1
trap "ouch; exit" 15
while :
do
  :
done
```

More usefully, one can trap on zero (shell exit), and delete temporary files. Those worried about `[t]csh` have no way of defining functions, let alone dealing with signals.

Shell Debugging

There are several useful debugging tools for shell scripts, all available via the `set` command.

`-e` Exit if command returns with non-zero status, and was not part of a conditional expression.

`-v` Print lines of script as they are executed.

`-x` Ditto, but after variable expansion.

Shell Stupidity

```
cd $somewhere
rm *
```

What happens if the `cd` fails?

Saner answers are

```
set -e
cd $somewhere
rm *
```

or

```
cd $somewhere && rm *
```

or

```
rm ${somewhere}/*
```

or

```
cd $somewhere || exit 2
rm *
```

Better yet, if you know the names of the files expected, don't be lazy, list them and avoid using `*`.

Stupid Quotes

```
$ cat l.sh
#!/bin/sh
for f in *
do
  ls $f
done
$ ./l.sh
correct
ls: hello not found
ls: world not found
l.sh
```

The directory contained three files, one of which was called ‘`hello world`’. For the second iteration, `$f` was set to ‘`hello world`’, and for the `ls` command the splitting into words happens after variable expansion, so `ls` saw two arguments, ‘`hello`’ and ‘`world`’. Writing
`ls "$f"`
would have been better.

UNIX fully supports spaces, `*`, `?`, `&`, `'`, etc. in file names. Many scripts, and many humans typing interactively, don’t, so it is a bad idea to use them.

POSIX says that portable filenames contain alphanumerics, and hyphen, underscore and dot. The first character must not be a hyphen, and the total length must be no more than 14 characters. The length might be a bit short, but the rest is sensible.

Not Shell Scripts

Getting Started

Shell scripts so far have started `#!/bin/sh` (or maybe `#!/bin/bash`) without much discussion of what this means.

When a file is marked executable, and the kernel wishes to execute it (presumably because it is the argument of one of the `exec()` functions), it examines the first few bytes to decide what to do with it.

It could be a binary file. Most current operating systems support four different types of these (32 bit and 64 bit, statically linked and dynamically linked). However, if the first two bytes are `#!` then the next word is taken to be the name of the interpreter to run, and then subsequent words are passed as arguments, then the name of the script passed as an argument, and finally any other arguments given. So

```
$ head -1 ./foo
#!/bin/whatsit  -a -z womble
$ ./foo bar
```

is equivalent to

```
$ /bin/whatsit -a -z womble ./foo bar
```

Universality

Provided the interpreter ignores the first line, and is happy to receive a command file as its last argument, this will work. As the Bourne shell treats any line starting with a # as a comment, and certainly accepts command files as the last argument, all is fine.

The `awk` command accepts a command file after a `-f` argument. So

```
$ awk '/^Energy/ {if ($2>Eold) {
   print "Warning! Energy increase from " Eold " to " $2;};
   Eold=$2;}' test2.dat
```

could have been written as

```
$ cat Echeck
#!/usr/bin/awk -f
/^Energy/ {
  if ($2>Eold) {
    print "Warning! Energy increase from " Eold " to " $2;
  };
  Eold=$2;
}
$ ./Echeck test2.dat
```

being equivalent to

```
/usr/bin/awk -f ./Echeck test2.dat
```

An Improvement

Not only does this allow one to space out the `awk` program more naturally, but it also frees one from concerns about quoting things from the shell's desire to expand variables and wild-cards. Now any expansion of arguments will be done by `awk`, and `awk` does not really do such things.

At this point writing `awk` scripts of about a dozen lines begins to seem sensible. However, with global variables, poor file handling, very poor abilities with multiple files, and running entirely interpreted, one should not get too carried away.

Ancient Checkers

In fixed format Fortran, characters after column 72 are ignored.

```
#!/bin/awk -f
# 17/5/95 MJR, after RWG
BEGIN { print "Scanning for lines longer than 72 chars."
        print "Ignoring FORTAN comments and trailing whitespace."}
(length > 72) && (substr($0,1,1) !~ /[cC]/ ) && (substr($0,73) ~ /[^ ]/ ) {
    printf "\nIn file %s\n",FILENAME
    printf "Line %8d:                                        Last valid char ->|\n",NR
    print }
END {print "Done"}
```

Not very pretty, and not precisely how I would write it now, but one written whilst I was a PhD student.

The isolated `print` will print the current line. The function `length` with no argument gives the length of the current line.

Others too

```
$ sed -e 's/^ *//' -e 's/  */ /g' -e 's/ *$//' < old > new
```

becomes

```
#!/usr/bin/sed -f
s/^ *//
s/  */ /g
s/ *$//
```

Another possibility is

```
#!/usr/local/shared/bin/gnuplot

set title "Simple Polynomials"
set key left
plot [0:2][0:6] x,x**2,x**3,x**4

pause -1
```

`perl` **and** `python`

These are two solutions to the problem of something being too complicated for `bash` and `awk`, and yet being unpleasant in C. Both handle multiple files, and, at a push, binary files. Both are byte-code compiled at run-time, making them much faster than the shell. Neither does spontaneous expansions of stars and dollars.

Unlike C, both support regular expressions and hashes, and have variables whose type is determined by context and contents, not definition.

One consequence of byte-compilation is that syntax errors throught the whole file get spotted before any of the file is executed. Interpreted languages, such as the shell, notice errors only when the interpreter reaches them.

The Importance of Parsing

Languages such as `perl` and `python` parse the whole file before executing any of it. This leads to slow start-up, but means that until the file is syntactically correct, none of it will run.

The Bourne shell parses lines and blocks immediately before execution. Syntax errors in the non-executed branches of conditionals will be spotted as execution passes by them.

The C shell parses and executes simultaneously. Syntax errors in the non-executed branches of conditionals are never spotted. So it takes longer to find the bugs.

```
#!/bin/sh
echo start
if false; then
  echo bar
#fi
echo end
```

```
#!/bin/csh -f
echo start
if ( 0 ) then
  echo bar
#endif
echo end
```

Both print ‘start’ and then complain about the lack of an end if. Replace ‘false’ by ‘true’ (and ‘0’ by ‘1’), and the `csh` script now runs perfectly and exits with a zero exit code, despite being syntactically incorrect. The behaviour of the `sh` script is unchanged.

The equivalent example in `perl` would simply fail to run at all.

perl

The older language is `perl`. It thus has a more traditional syntax, is more mature (evolving less rapidly), and is very widespread. If a machine does not have at least perl version 5.0 installed on it, it probably has a large number of other historically interesting defects too. The disadvantage of perl is that it uses far too many unmemorable bits of shorthand, leading to compact, unreadable, code.

Perl first appeared in 1987. Perl 5 appeared in 1994. The current version is perl 5.10, which has added several features since 5.0, but will still run any 5.0 scripts.

Perl 6 is planned, and will not be backwards compatible with perl 5. It has been in development since 2000, and no release date is yet (2009) set.

By Example

```
#!/usr/bin/perl

while(<>){
  if (/^Energy/) {
    ($junk,$E)=split;
    if ($E>$Eold) {
      print "Warning! Energy increase from ".$Eold." to ".$E."\n";
    }
    $Eold=$E;
  }
}
```

The names of all scalar variables must start with `$`.

Filehandles don't start with `$`. The null filehandle refers to `stdin`, unless there is an extra command line argument, in which case it is assumed to be a file and associated with the null filehandle.

The Lesson

The `< >` operator reads a line from the given filehandle, and places it in `$_` (unless it is assigned to a variable).

The `split` operator splits up something (by default `$_`) on something (by default whitespace) and returns an array of strings.

The parentheses make an array, and the assignment happens element by element, with an excess elements on the right discarded.

The `print` command takes a single argument, or comma-separated arguments. Here the dot operator (string concatenation) is used to produce a single argument. No new line will be produced automatically.

Braces must follow `if` (and `for`, `while`, etc.) even if just a single statement (or even no statement) completes the construction.

Fast!

Recall the earlier test which showed a shell loop using `expr` to increment a counter managing one cycle per 2ms, and using the `$(( ))$` syntax a more reasonable 35μs per cycle?

```
$ /usr/bin/time perl -e 'for($i=0;$i<10000000;$i++){}'
  0.72 real   0.71 user  0.00 sys
```

That is about 70ns per cycle. Still a lot slower than a truely compiled language, such as C or Fortran, which would manage 3ns or better, but a big improvement.

Where does `perl` excel?

It is possible to produce one-line perl scripts, and sometimes they can be very useful.

```
$ perl -e 'print "-"x80 ."\n";'
```

(print precisely 80 hyphens)

```
$ perl -e 'print crypt("password","SL")."\n";'
```

(Print the result of crypt()ing password with given salt.)

It also has support for networking – opening sockets, looking up addresses, etc.

It supports loadable libraries (called 'modules'), which can contain compiled C code.

Other perlisms

```
$ awk '/^#/ {next} {t+=$2; c++;} END {print t/c;}' test.dat
```

```
while(<>){
  next if (/^#/);
  ($junk,$value)=split;
  $t+=$value;
  $c++;
}
print $t/$c,"\n";
```

(Note that the `if` can follow the statement which depends on it.)

Using Files

Files are opened with the open command:

```
open(FRED,"<in.dat");
open(BILL,">out.dat") || die "Failed to open output";


print BILL "And the result is...\n";
close(BILL);
```

Note no comma in `print` (or `printf`) after the file handle, unlike C.

Note that filehandles are capitalised by convention, and do not start with a `$`. Missing the `$` from a scalar variable name causes it to be interpreted as a filehandle or label. This often remains syntactically correct, so no error is given, and the final result is unexpected.

PostScript Bounding Boxes Again

```
open(IN,"<".$infile) || die "Cannot open ".$infile;

while(<IN>){last if /^%%BoundingBox:/;}

@bbox=split;

if ($bbox[1] eq '(atend)'){while(<IN>){@bbox=split if /^%%BoundingBox:/;}}
shift(@bbox);
```

Note that the names of arrays start with a `@`, whereas the name of an array element, being a scalar, starts with `$` as usual. The `shift` command moves all elements of the array down one.

Hashes also exist (see previous lecture under `awk`). Their names are prefixed with `%`, and their index (or key) is enclosed by `{}` not `[]`.

Speed Again

Having created a largish text file of 39,650 lines and just under 2MB, it was read in in three ways:

```
while read x
do
  :
done
```

The shell, which took 1.1s.

```
perl -e 'while(<>){;}'
```

Perl, which took 0.02s.

```
grep '^%%EOF'
```

Grep (there were seven matches), which took 0.008s. (In all cases the file and the command were in cache, not on disk.)

Mandelbrot in `perl`

```
#!/usr/bin/perl

sub mand {
  my $zr,$zi,$tmp,$i;

  ($z0r,$z0i,$n)=@_;

  $zr=$z0r; $zi=$z0i;

  for($i=1;$i<$n;$i++){
    $tmp=$zr*$zr-$zi*$zi+$z0r;
    $zi=2*$zr*$zi+$z0i;
    $zr=$tmp;
    last if (($zr*$zr+$zi*$zi)>16);
  }

  int(512-512*log($i)/log($n));
}
```

```
sub ppm_write {
  my ($i,$j,$resx,$resy,$name,@set);
  $name=shift(@_);
  @set=@_;

  $resx=$#set+1;
  $resy=$#{$set[1]}+1;

  open(OUT,">".$name);
  print OUT "P6\n";
  print OUT $resx," ",$resy,"\n";
  print OUT "255\n";

  for $i (0 .. $resy-1){
    for $j (0 .. $resx-1){
      print OUT pack("C3",@{$set[$j][$i]});
    }
  }
}
```

```
print "Please input resolution\n";
$res=<>;

$pallette[0]=[(0,0,0)];
for $i (1 .. 256){
  $pallette[$i]=[(256-$i,256-$i,$i)];
}
for $i (1 .. 256){
  $pallette[$i+256]=[($i,0,256-$i)];
}

for $ix (0 .. $res-1){
  $zr=-2+$ix*3.0/($res-1);
  for $iy (0 .. $res-1) {
    $zi=-1.5+$iy*3.0/($res-1);
    $set[$ix][$iy]=[ @{$pallette[&mand($zr,$zi,1024)]} ];
  }
}

do ppm_write("p.pnm",@set);
```

Comments on Mandelbrot

No support for complex numbers in `perl`.

Multidimensional arrays are awkward: they are really arrays of arrays. However, this is little worse than C! What is worse than C is that they behave differently depending on context, so a hideous collection of `@`, `[` and `{` can be needed to perform the correct sort of cast, and even then there are worries about whether two arrays are copies of each other, or pointers to the same memory location.

Binary I/O is no problem via `pack` and `unpack`. Nor is having a null in the middle of a string, and then passing that string to `print`.

Arguments to functions get flattened and passed in a single list object `@_`. It can be hard to rescue an array of unknown length from the resulting mess.

It is slow. The standard example used in the parallelism lecture took about 190s, compared to 5s in Fortran.

python

Python is the other major byte-compiled cross-platform scripting language. It is rather more modern than `perl`, with python 1.0 being released in 1994 (the same year as perl 5.0). In general it is marginally faster than `perl`.

Python version 3 was released in December 2008, and deliberately breaks backwards compatibility in order to fix a few design problems.

So most people would be better off with 2.6, as more software is written for it. However, as 2.6 was released in October 2008, not many people will be using it yet. Version 2.5 was released in September 2006, so will be more wide-spread, but some current Linux distributions, such as SLES10 SP2, are still using 2.4 (released November 2004).

Extensible Pythons

Python is easier to extend than `perl`, and many, many packages add functionality. One of interest to the scientific community is `NumPy`, which adds multidimensional arrays, LAPACK and FFTs. Because the LAPACK and FFT parts are written in a proper compiled language, not python, they are fast. The package `SciPy` builds on `NumPy` and adds yet more features, forming a poor man's MatLab.

Extension Problems

The problem with these extensions is compatibility and availability.

Three different extensions have added multidimensional arrays to python (NumPy, Numeric and Numarray). It is now clear that NumPy is the winner, but it has only really existed since about 2005, whereas Numeric appeared in 1995.

Because NumPy is not part of the standard core of python, there is no guarantee than any given python installation will have it. Again, SLES10 SP2 has no NumPy package, although it could do so as python 2.4 is the minimum python version that NumPy is intended to work on.

NumPy does not yet support python 3.

MacOS 10.5 has python 2.5, but not NumPy or Numeric.

The `python` Language

The `python` language is much more object oriented than `perl`, and it does not use braces to delimit blocks of code, but rather relies entirely on indentation. People either love or hate the result.

The good news is that it supports complex variables. The bad is the poor support for binary I/O and arrays.

We start with a rather unusual example for python, the Mandelbrot set again.

Mandelbrot, Part I.py

```
#!/usr/bin/python
from Numeric import *

def mand(z0,n):
  z=z0
  for i in range(1,n):
    z=z*z+z0
    if abs(z)>4 : break

  return 255-255*(log(i)/log(n))

def pgm_write(set,name):
  [width,height]=set.shape

  pgm=open(name,mode='w')
  print >>pgm,"P2"
  print >>pgm,width,height,"\n255"

  for iy in range(0,height):
    for ix in range(0,width):
      print >>pgm,set[ix,iy]

  pgm.close()
```

Mandelbrot, Part II.py

```
print "Please input resolution"
res=int(raw_input())

set=zeros([res,res],Int)

for ix in range(0,res):
  x=-2.+ix*3./(res-1)
  for iy in range(0,res):
    y=-1.5+iy*3./(res-1)
    set[ix,iy]=mand(complex(x,y),1024)

pgm_write(set,"p.ppm")
```

The Lessons

A statement which expects a block ends with a colon, and has the block indented.

The `for` loop iterates over a list. The `range` operator provides one, which excludes the final value.

Objects have methods. If `pgm` is a filehandle, then `pgm.close()` exists. If `set` is an array, then `set.shape` exists and gives its dimensions.

It is slow! In Fortran this code ran in $6\frac{1}{2}$s. In Python it takes over 100s, even with the I/O section removed.

Fortran without I/O took 5s, so the Python penalty is a factor of over 20 in this case. However, for this Python is faster than `perl`.

A PostScript version was only marginally worse at 120s. (Though PostScript is only single-precision.)

Some Real Examples: Graphics Conversion

The `eps2bmp` script in TCM is a Bourne shell script with about 180 non-comment lines. Ideas for extracting bounding boxes from PostScript files have already been covered, and the script essentially constructs a command such as

```
cat x.eps | gs -sDEVICE=ppmraw -q | pnmscale 0.5 | pnmtopng > x.png
```

It does a couple of tricks not already covered.

Note it is common in UNIX for a command to perform one, simple, operation. So scaling a bitmap (with antialiasing) and converting to png format are separate commands here.

Here!

```
{
cat << EOH
%!
/showpage {} def
/setpagedevice {pop} def
$llx neg $lly neg translate
EOH

cat $infile
echo showpage
} | gs ...
```

The syntax `<<label ... label` creates a 'here document.' This section of the script is fed to the preceding command as `stdin`. The extra braces ensure that `stdout` from the two `cat`s and the `echo` all get piped into `gs`.

These 'here documents' do not exist in `(t)csh`. They can be useful for getting multiple lines of input into a command.

Variable names are interpreted in them, so `$llx` is replaced by the value of the shell variable `$llx`. One can suppress this.

Booklet Printing

These booklets are the result of a print command such as

```
psbook out.ps | psnup -2 | duplex -land -Ppsc
```

The `duplex` command itself merely needs to take its input and prefix a few lines to turn on duplexing in PostScript. Thus it is something like

duplex

```
#!/bin/sh
out="cat"
tumble=false
if [ "$1" = -land ]
then
 tumble=true
 shift
fi

case x$1 in
 x-P*)
  out="lpr $1"
  shift
  ;;
esac

{ echo "%!"
  echo '<< /Duplex true'
  echo "   /Tumble $tumble"
  echo '>> setpagedevice'
  echo '/setpagedevice {pop} bind def'
  cat $1
} | $out
```

Error Conversion

```
$ cat data
Force 1    3.2567(2)
Force 2   26.783(4)
Force 3  134.5(1)
Force 4    0.00012(3)
$ ./err.pl < data
3.2567 +/- 0.0002
26.783 +/- 0.004
134.5 +/- 0.1
0.00012 +/- 0.00003
```

There follow two ways of doing this, neither necessarily perfect, but to show that this is doable in a few lines of `perl` or `python`.

Error Conversion in `perl`

```
#!/usr/bin/perl

while(<>){

    m/([-.0-9]*)\(([0-9])\)/ || next;

    $value=$1;
    $err_digit=$2;

    $value =~ m/\.[0-9]*/;
    $err='0.'.'0' x (length($&)-2).$err_digit;

    print $value,' +/- ',$err,"\n";
}
```

The match operator, `m`, sets `$1`, `$2`, etc., to the bracketed subexpressions in the extended regular expression given. It is an ERE, not a BRE, so literal brackets have to be preceded by backslashes.

The `=~` operator makes `m` operate on the given variable, but does not change the variable. It does set `$&` to the string matched.

Error Conversion in `python`

```
#!/usr/bin/python

import sys
import re

r1=re.compile(r'([-.0-9]*)\(([0-9])\)')
r2=re.compile(r'\.[0-9]*')

for line in sys.stdin:
  try:
    (value,err_digit) = r1.findall(line)[0]
  except IndexError:
    continue

  tmp=r2.search(value)
  zeros=tmp.end()-tmp.start()-2
  err='0.'+'0'*zeros+err_digit
  print value,'+/-',err
```

Two standard, but not automatically loaded, modules need including, `re` and `sys`.

Prefixing a string constant by an `r` causes it to be treated as a raw string, without python interpretting backslashes etc.

Regular expressions are compiled and assigned to regular expression objects, which then have methods such as `findall`.

Control Freaks

```
#!/bin/bash

normal_exit(){
  cp ${tmpdir}/output ${HOME}/output
  rm ${tmpdir}/*.tmp; exit
}

cleanup(){
  kill -TERM $prog
  sleep 5
  normal_exit
}

trap cleanup XCPU
tmpdir=/scratch/spqr1
./a.out >& ${tmpdir}/output &
prog=$!
wait $prog
normal_exit
```

Control Comments

The above might be the nucleus of a script for controlling a program under a queueing system. When the program ends, files get copied out of `$tmpdir` (for use on nasty computers which delete a job's fast scratch space as soon as the queueing slot ends), and should the queueing system send the XCPU signal, the script will send the TERM signal to its child program, wait a few seconds, and then try copying the output file.

Note that `[t]csh` users stand no chance: no shell functions, no signal handling, no `wait` command, no ability to find the process ID of the last thing launched via a nice variable like `$!`. However, `perl` and `python` users would be able to do something similar.

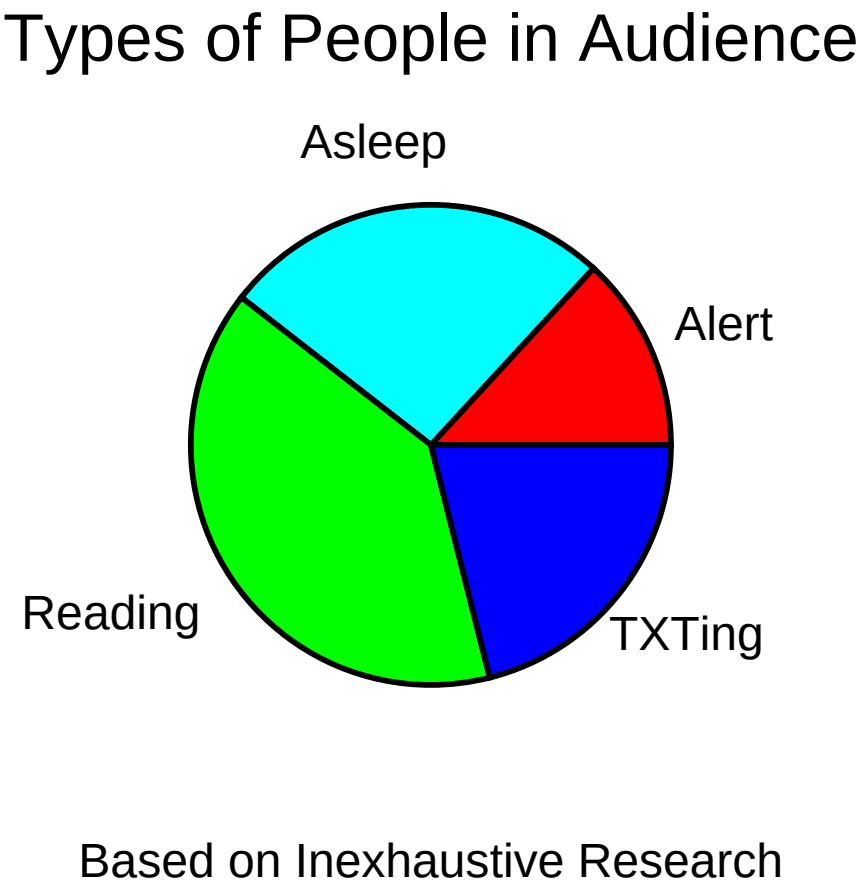
`awk` Fun

```
$ cat demo.dat
# A commented data file

Title   Types of People in Audience
Caption Based on Inexhaustive Research

5 Alert  col= 1 0 0
10 Asleep col= 0 1 1
15 Reading col= 0 1 0
8 TXTing  col= 0 0 1
$ ./pie.awk < demo.dat > demo.eps
$ gv demo.eps
```

Useful, because combined with `eps2bmp` one could automatically produce a pie chart for a WWW page in `gif` format, with antialiasing.



pie.awk

```
#!/usr/bin/awk -f

# Create a pie chart in EPS from a two column list

BEGIN{n=1;}

/^#/ {next;}    # Skip comments
/^ *$/ {next;}  # Skip blanks

/^Title/ {title=$0; sub(/^Title[        ]*/,"",title); next;}
/^Caption/ {caption=$0; sub(/^Caption[  ]*/,"",caption); next;}

{label[n]=$2;data[n]=$1;n++;total+=$1;}
($3 == "col=") {datacol[n-1]=$4 " " $5 " " $6;}

END {

# First define some segment colours

 segcol[0]="1 0 0"; # red
 segcol[1]="0 0 1"; # blue
 segcol[2]="1 1 0"; # yellow
 segcol[3]="1 0 1"; # magenta
 segcol[4]="0 1 0"; # green
 segcol[5]="0 1 1"; # cyan
 nsegcol=6;
```

```
# Avoid first and last segments being same colour
 if ((n-1)%nsegcol==1) {nsegcol--;}

 print \
 "%!PS-Adobe-3.0 EPSF-3.0\n" \
 "%%Creator: MJR script\n" \
 "%%DocumentFonts: Helvetica\n" \
 "%%BoundingBox: -100 -100 100 100\n" \
 "%%Pages: 1\n" \
 "%%EndComments\n" \
 "% angle1 angle2 label segment\n" \
 "/segment {\n" \
 " /lab exch def\n" \
 " exch\n" \
 " /start exch def\n" \
 " /end exch def\n" \
 "% draw it\n" \
 " newpath\n" \
 " 0 0 moveto\n" \
 " 0 0 50 start end arc\n" \
 " closepath\n" \
 " gsave fill grestore\n" \
 " 0 0 0 setrgbcolor stroke\n" \
 "% add label\n" \
 " start end add 2 div dup dup /angle exch def\n" \
 "% angle at centre\n" \
 " cos 60 mul /tx exch def sin 60 mul /ty exch def\n" \
 "% posn of centre of label, now to find width and height of label (tw and th)\n" \
 "% lab stringwidth /th exch def\n" \
 " newpath 0 0 moveto lab false charpath flattenpath pathbbox closepath /th exch def /tw exch def pop pop\n" \
```

```
" angle 2 div sin tw mul neg tx add\n" \
"% corrected x posn left on stack\n" \
" angle sin 2 div -.5 add th mul ty add\n" \
" moveto\n" \
" 0 0 0 setrgbcolor\n" \
" lab show\n" \
"} def\n" \
"/label {\n" \
" /txt exch def\n" \
" /ly exch def\n" \
" /lx exch def\n" \
" lx 0 lt\n" \
" { txt stringwidth pop neg lx add /lx exch def } if\n" \
" 0 0 0 setrgbcolor\n" \
" lx ly moveto\n" \
" txt show\n" \
"} def \n" \
"\n" \
"%%EndProlog\n" \
"%%Page: 1 1\n" \
"\n" \
"1.2 setlinewidth\n";

if (title != "") {
  print "/Helvetica findfont 14 scalefont setfont";
  print "(" title ") stringwidth";
  print "pop 2 div neg 80 moveto";
  print "(" title ") show";
}

print "/Helvetica findfont 10 scalefont setfont\n";
```

```
 if (caption != "") {
   print "(" caption ") stringwidth";
   print "pop 2 div neg -90 moveto";
   print "(" caption ")show";
 }

 pos=0;
 colour=0;
 for (i=1;i<n;i++) {
  segsize=data[i]*360/total;
  if (segsize>0.75){        # Suppress label and print black if < .75 degree
    newpos=pos+segsize;
    if (datacol[i] != ""){
      print datacol[i] " setrgbcolor";
    }
    else{
      print segcol[colour] " setrgbcolor";
      colour++;
      if (colour >= nsegcol) {colour=0;};
    }
    print  pos " " newpos " (" label[i] ") segment";
  } else {
    print "0 0 0 setrgbcolor";
    print  pos " " newpos " ( ) segment";
  }

  pos=newpos;
}
print "showpage";
}
```

Comments on `pie.awk`

At that length, it's probably time we learnt `perl` or `python`.

PostScript output is easy from any language which can print ASCII text.

PostScript deals nicely with circular arcs, text in neat scalable fonts, and colour. If one wanted a bitmap, it is simplest to write in PostScript, and then feed to `ghostscript` in some form.

`perl` **vs** `python` **again**

If I knew no Fortran, `python` would look attractive for its support for complex numbers, and its ability to interface to fast FFT and Lapack libraries.

If I loved C++ or Java for their 'modern' object orientation and try/except style of exception handling, again I would find `python` attractive.

Given that I do know Fortran, and don't much love C++/Java, `python` looks less attractive. Its difficulties with binary I/O, and the need to `import` things to get more than very basic functionality, make the traditional `perl` look quite attractive still.

Both are used quite extensively to drive CGI-based web pages.

Finding deficiencies in either is worryingly easy...