UNIX shell scripting for high-throughput Phylogenetics.

Aim: To use the command line shell environment to execute batch jobs and to create a discovery pipeline for bioinformatics.

Introduction to the shell.

The UNIX/LINUX shell scripting environment is one of the most important and powerful ways of using the computer. When you log onto a UNIX machine, you enter an environment called the shell. This is the part of the UNIX system that controls the resources of the UNIX Operating system. There are a number of different flavours of this environment. The original shell **sh**, was written by Steve Bourne and is known as the Bourne shell. Other common shells include the C shell **csh**, the enhanced C shell **tcsh**, the Korn shell **ksh**, and the “Bourne Again Shell” **bash**. All of these various shells offer enhancements to the Bourne shell. For Linux, the default shell when you log in is the **bash** shell. You will be presented with a ‘prompt’ and at the prompt you will type commands. In this practical, the prompt is represented by a dollar sign ($), do not type the dollar sign.

Commands:

Ordinarily, you will enter a single command on the command line. E.g.:

```
$ date
```

This will print the current date and time. However, you may also group commands together by separating them using a semi-colon. For instance:

```
$ date; ls
```

This will execute the *date* command, when the date program exits, the ls program will execute. The ls command lists the contents of the current directory.

Using wildcards to specify files

The shell gives you a way to abbreviate files. This can be achieved using the * wildcard. For instance, if you wish to list all the files in a directory that end in the extension .fasta, you could issue the command:

```
$ ls *.fasta
```

Other alternatives include:

```
$ ls bovine*      //all files beginning with the string bovine
$ ls *paup*       //list all files containing the string paup
                  //anywhere.
```

The question mark “?” matches any single character:

```
$ ls bovin?       //Matches “bovine”, but not bovine.paup.
$ ls bovin*       //Matches “bovine” and also bovine.paup.
$ ls bovin[Ee]    //Matches “bovine” and “bovinE”.
$ ls bovin[a-f]//Matches “bovina”, “bovinb”… “bovinf”.
```

The following is a brief summary of the most frequently-used unix commands. Type each command and become familiar with its usage.

Commands	Explanation
alias	permits redefinition of an existing command with another string
alias listlong=’ls –la’	Define the command ‘listlong’ as an alias of the command ls -la
apropos	lists all the man entries relating to a topic (same as man -k)
apropos date	Show all manual entries that contain the word date.
cat	concatenates and displays files
cat myfile	displays myfile on the screen
cat one.seq two.seq > both.seq	writes two files into one
cd	changes current directory cd /etc cd .. cd ../sub2
cmp	Compares two files and prints the character and line number where they first differ.
cmp newfile oldfile	(see also diff)
cp	copies files
cp file1 file2	Copy the contents of file1 into a new file called file2
cp /tmp/sequence.fas .	Not the ‘dot’ at the end, this means ‘here’. Copy the file called ‘sequence.fas’, which is located in the directory ‘/tmp’ to here.
date	displays current date and time
echo these words	Echo prints out the command line parameters (i.e. “these words”).
exit	leaves the current shell (same as ^d) usually = logout
find	searches the directory tree find / -name lostfile
finger	tells you who is logged on (see also w and who)
grep	searches a file for a string: grep word file grep 'two words' file
head	prints the first few (default = 10) lines of a file: head oddfile
head -5 msgfile	(displays first five lines)
history	displays last several commands used
!!	re-executes the last command
!51	executes command 51 in the history list
jobs	lists background processes (created with ^z or bg)
jobs -l	(el not one) includes the pid (process id number)
kill	stops a process (use ps aux | grep $USER to find your processes)
kill 2986	kills off the process with pid 2986
kill -9 2986	definitely kills off pid 2986
ls	lists files in a directory
ls -alF /usr/local/bin	lists 1) all files in 2) long format 3) identifies directories / executable files * and symbolic links @ in the directory /usr/users/bin
more	Displays a file one screenful at a time:
more longfile	Displays longfile one screenful at a time

man	Gives manual information on a topic
man kill	Gives manual information on the ‘kill’ command
man ls	Gives manual information on the ‘ls’ command.
mkdir	Creates a new subdirectory mkdir subd
mv	(move) renames a file (or dir.)
mv file1 file2	Renames “file1”. Its new name is “file2”.
mv file1 /subd/file1	Moves “file1” to a subdirectory called subd and in this subdirectory, its name is “file1”.
passwd	Invokes a password changing program
rm	Removes/deletes a file. -i option advised if wildcards in use:
rm -i *.seq	Removes (in interactive mode) all files with the .seq extension.
rmdir	Removes a directory - you must delete all the files in it first
tail	Displays last few lines of a file (see head)
unalias	Destroys a previously set alias (which see)
w	(who) displays information about logged in users (see finger)
whoami	For those having an identity crisis
|	This is a pipe and can be used to redirect the output from one command into another command
head –100 myfile | tail –10	This series of commands extracts the top 100 lines from ‘myfile’ and then extracts the last 10 lines from this output. The result is that lines 90 to 100 are printed to the screen.
head –100 myfile | tail –10 | more	Pipes the output to the program ‘more’

NOTE: You can use the autocomplete facility in UNIX by hitting the <tab> key after typing a shorter version of a command. This only works if the abbreviated version is unique to that command. e.g.:

```
$ clus<TAB>
```

produces

```
$ clustalw
```

Redirect (>), Append (>>) and Pipe (|)

In UNIX, the output is usually directed, by default to the screen. This is not necessarily always the most desirable place to put the output. You may wish to redirect the output to a file, so that you may look at the results at a later stage, or you may wish to use the output from one program as the input to another program. List the contents of a directory, but instead of putting the output to the screen, redirect the output to a file:

```
$ ls /usr/local/bin > tmp
```

This puts the contents of the command 'ls' into a file called "tmp". Take a look at the contents of the file tmp, using the command **more**.

```
$ more tmp
```

(**more** displays the contents of a file, one page at a time. You can use the space bar to advance the display by one page and you can use the 'q' key to quit.)

You may append some information to the end of a file by using the >> operator. This operator appends without overwriting.

```
$ ls /usr/bin >> tmp
```

This appends the contents of the directory "/usr/bin" to the file "tmp". However, if you wish to list the contents of a directory and view the results one page at a time, then you can **pipe** the contents of the ls command straight into the more command:

```
$ ls /usr/local/bin | more
```

The vertical bar is called a pipe and it can be used to redirect output from one command so that it can be used by another command.

grep

The grep command is probably one of the most useful of the unix commands. Grep stands for "General Regular Expression Parser" and it is a UNIX command that is very useful for finding lines in a file that contain a string of interest. The general format of a grep command is:

```
$ grep PATTERN file
```

So, for instance, if you have a file full of fasta-formatted sequences and you just want to look at the names, you could use the command:

```
$ grep ">" sequencefile.fas
```

Because the lines that contain the sequence names in a fasta-formatted file have the unique identifier > it is possible to just look at those lines.

Shell Scripts

The UNIX shell allows the user to compose files called shell scripts. These files usually contain multiple commands that are executed by the Operating System.

A shell script is created using a text editor and then executed by the operating system. We will use the 'emacs' text editor. You may start this text editor by typing:

```
$ emacs
```

or

```
$ emacs script.sh
```

The first command starts an empty emacs session, the second command starts an emacs session with a file called 'script.sh' (this file will be created if it does not already exist). You may exit emacs by typing a series of 'escape' keys. This key combination is:

```
<cntrl>x<cntrl>c
```

In other words, you hold down the control key and press x, then hold down the control key and press c. The first line of the file should contain the text:

```
!/bin/bash
```

This tells the operating system that this is a script and that it should be executed using the bash shell.

Providing arguments to shell programs.

The simplest and often the most effective way to use the UNIX operating system is to write short scripts that perform repetitive tasks. These shell scripts are part of the reason why UNIX is so popular. When you have a variety of scripts assembled that perform various tasks, then you will have the beginnings of a phylogenomics infrastructure.

You may write a shell script for performing a general function (for instance calling a multiple alignment program to align a file of DNA sequences), but you may wish to provide the script with different input values each time you run it (different sequence file name, different gap opening penalties, etc). Therefore, it is necessary to be able to pass arguments to the shell script. You can provide arguments to a shell program by specifying them on the command line. In other words, you type the name of the script, followed by a number of input parameters. When you execute a shell script, shell *variables* are automatically set to whatever is specified on the command line. These variables are referred to as positional parameters.

The parameters $1, $2, $3, $4, $5 refer to the first, second, third, fourth and fifth arguments passed to the script. The parameter $0 refers to the name of the shell program. The parameter $# is the *number* of arguments passed to the script.

```
cmd   arg1  arg2  arg3  arg4  arg5
|     |     |     |     |     |
$0    $1    $2    $3    $4    $5
```

So, for instance the following shell script (copy this into a file) prints out the command line parameters to the screen.

```
echo $0
echo $1
echo $2
echo $3
echo $4
echo $5
```

If we call this script ShowArgs.sh and we execute this script, like this:

```
$ sh ShowArgs.sh these are the arguments so there!
```

The output will be:

```
            show_args
            these
            are
            the
            arguments
            so
```

i.e. the output is the first six command-line parameters.

The ‘if’ command

A simple kind of script program allows conditional execution depending on whether some question is true. In the shell, the **if** operator provides simple program control through simple branching. The general form of the **if** command is:

if *command*
      then *command*
fi

The command following the **if** is executed. If it completes successfully, the command following the **then** is executed. The **fi** (if spelled backwards) marks the end of the **if** structure. UNIX commands provide an exit status when they complete. By default the return value of true is zero, otherwise a nonzero return value is returned. In the above example a zero return value for the **if** statement results in the command after **then** being executed. You may write this script in a file or type it on the command line (the hashes are just my comments):

```
if
clustalw sequences.fas        #if this is true (if it runs properly)
then echo YIPEE!!!!           #then run this command
fi
```

Please note the behaviour of the operating system for this operation.

The ‘test’ command

Here we can test whether or not something is true. For instance we can test whether or not a file exists:

```
if
test –f sequences.blah
then echo YIPEEE!
else echo AWWWWW!
fi
```

NOTE: use ‘man test’ to see what other options are available for test.

The ‘for’ command

The **for** loop executes a command list once for each member of a list. The basic format is

```
for i
      in list
do
      commands
done
```

What does this script do?

```
            for i in 1 2 3 4 5 6 7 8 9
                  do
                  echo “Hello, World”
                  done
```

If you want to specify this kind of thing at the command line, you might write a shell script like this:

```
            for i in $*
                  do
                  echo $i
                  done
```

In English, this shell script is effectively saying “for every item that you type at the command line (that is what the `$*` means), I will echo its contents to the screen”. Therefore, if you write this script and call it ***echoscript.sh*** and run the script using this command:

```
sh echoscript.sh CAT MoUsE human /k/hldu
```

It will produce the output:

```
CAT
MoUsE
human
/k/hldu
```

Exercise

Background: Frequently, it is necessary to perform many different operations on many different files. It is also necessary to create 'pipelines' in order to quickly and easily analyse data. These exercises are designed to create tools that can be used to perform large scale analyses without user intervention. We shall run some datasets through three programs, however, we will not do it manually, rather we shall write a script to do this automatically.

1. Write a simple shell script that prints out all the lines that contain the sequence names. You know how to do this already.

2. Write a shell script that calls clustalw and aligns all the sequences in every one of the fasta-formatted files. The general command line parameter that you should use to call clustalw is:

```
clustalw -infile=file.in -outfile=file.out -output=NEXUS
```

3. Finally, append the following text to the NEXUS-formatted file (this text contains a "PAUP block" which can be read by the PAUP program).

```
Begin paup;
      Set criterion=distance;
      Dset distance=logdet;
      Savedist file=$i.dist format=phylip;
      Nj;
      Savetrees file=$i.tre;
      Quit;
Endblock;
```

N.B.: when I have used the **$i** notation in part 3, I intend this to be a variable that can change for each file.

HINT: Remember the echo command? You can use this to append text to a file.

4. The shell script must then execute the PAUP program with this completed file as the input for PAUP.
5. Print the distance matrices to the screen.
6. Next get P4 to print the resulting tree to the screen.