

BASH Scripting

Dr. Leigh Brookshaw

April 18, 2006

Abstract

The UNIX “shell” is more than a command interpreter. It has a programming language of its own that can be used to write shell programs for performing various tasks that cannot be performed by any existing command. A shell program, commonly known as a **shell script**, consists of shell commands to be executed by a shell and is stored in a regular UNIX file.

This document introduces the default GNU/Linux shell and extends the simple interactive shell commands that all users of UNIX are familiar with to complex shell scripts.

After reading through this document and attempting all the exercises you should be able to be able to write your own shell scripts and understand and modify the many shell scripts that control your UNIX system.

Table of Contents

1.1	Introduction	3
1.2	Shell Basics	4
1.2.1	Globbing Pathnames	4
1.2.2	Redirection	6
1.2.3	Pipes	10
1.2.4	Command Lists	12
1.2.5	Here Documents	16
1.2.6	Command Substitution	16
1.2.7	Job Control	17
1.2.8	Variables	20
1.2.9	Quoting	26

1.3	Shell Programming	28
1.3.1	Shell Scripts	28
1.3.2	Special Variables	30
1.3.3	Arithmetic Expressions	31
1.3.4	Conditional Expressions	33
1.3.5	Control Structures	36
1.3.6	Functions	54
1.4	Script Examples and Exercises	57
1.5	Further Reading and References	65

1.1 Introduction

The shell is the part of UNIX that is most visible to the user. It receives and interprets the commands entered by the user. To do anything on UNIX the user must give the shell a command. If the command requires a program to be run, the shell requests that the kernel run the program.

The shell is a program like any other running on the UNIX system. The UNIX login procedure will guarantee that after you login you will own a single process—the login shell. The login shell will have standard input, output and error assigned to your terminal.

There are two major parts of a shell. The first is the interpreter. The interpreter reads your commands and works with the kernel to execute them. The second part of the shell is a programming capability that allows the shell to execute a *shell script* or *shell program*. A shell script is a file that contains shell commands that perform a useful function.

The common commands used by a user are only a small subset of the commands available to the shell. The shell recognises variables, conditional expressions, control structures—all the elements of a sophisticated programming language.

There are a number of different shells that can be used under UNIX:

Bourne The Bourne shell, developed by Steve Bourne, is the oldest shell. Because it is the oldest it is the most basic of the shells available today. Its interactive features are not as user friendly as more modern shells.

C The C shell was developed by Bill Joy and received its name from the fact that its commands were supposed to look like C statements. The C shell has many features that make it a better interactive shell to the Bourne shell. It was common for users to write scripts in the Bourne Shell even if they preferred

the C shell for interactive use.

T An extension to the C shell that incorporated filename completion and command-line editing. Important interactive features.

Korn The Korn shell, developed by David Korn, is an extension of the Bourne shell that incorporated many of the features of the C shell that made it popular as an interactive command interpreter.

Bash The “Bourne again shell” is an extension of the Bourne shell that incorporates useful features of the C shell and the Korn shell. Under GNU/Linux the Bash shell is the default shell.

1.2 Shell Basics

1.2.1 Globbing Pathnames

Globbing is the operation that expands a wildcard pattern into the list of pathnames matching the pattern. A wildcard pattern is a string that contains one of the following ? * [].

Each filename must be unique. At the same time, we often need to work with a group of files. A wildcard pattern can be used to match a group of filenames.

Matching Zero or More Characters

The asterisk ‘*’ is used to match zero or more characters in the filename.

Example 1.1:

<code>*</code>	match every file (see below)
<code>*.jpeg</code>	match all files with a jpeg suffix
<code>*.*</code>	match all files that contain a period
<code>m*.gif</code>	match all files that start with m and have a gif suffix

If a filename starts with a '.', this character must be matched explicitly. This means that the command `'rm *'` will not remove the file `.muttrc`—but the command `'rm *.*'` will remove it.

Matching Any Single Character

The wildcard character `?` will match any single character.

Example 1.2: To list only the files `pic1.jpg`, `pic2.jpg`, to `pic9.jpg` then use the following command:

```
prompt: ls pic?.jpg
pic1.jpg      pic4.jpg      pic7.jpg
pic2.jpg      pic5.jpg      pic8.jpg
pic3.jpg      pic6.jpg      pic9.jpg
```

The `?` can appear more than once in a string. Each time it appears it will match one and only one character in the filename.

Matching a Single Character from a Set

The wildcard construct `[...]` defines a set of characters. Like the single-character wildcard, a set matches only one character in the filename.

Example 1.3:

<code>c[aeiou]t</code>	matches <code>cat</code> , <code>cet</code> , <code>cit</code> , <code>cot</code> , and <code>cut</code> only.
<code>[A-Z]*</code>	matches any file that starts with a capital letter.
<code>[A-Za-z]*</code>	matches any file that starts with an alphabetic character.
<code>c[!aeiou]t</code>	matches all files that begin with <code>c</code> and end in <code>t</code> and only have a consonant for the middle letter.

The `-` character is used to specify a range of characters. This range is in the ASCII sense—that is, the range is calculated from the ASCII table.

A leading `!` negates the set. That is, the following characters are not matched in the filename all others are.

1.2.2 Redirection

UNIX defines three standard streams that are used by commands—**standard input**, **standard output**, and **standard error**. These are identical to the standard input, output and error streams of the programming language C. UNIX assigns a descriptor to each of the streams—standard input is 0, standard output is 1, and standard error is 2.

When you create an Xterm the standard input is preassigned to come from the keyboard, and the standard output and error are sent to the Xterm to be displayed. Whenever necessary, we can change these default assignments temporarily using redirection.

Redirecting Input

Standard input can be redirected from the keyboard to any text file. The input redirection operator is the `<` character.

The syntax for redirecting standard input is

```
command 0< file.txt
```

or

```
command < file.txt
```

The first form explicitly specifies the standard input descriptor. If the descriptor is omitted standard input is assumed.

Redirecting Output

Standard output can be redirected from the terminal to any text file. The output redirection operator is the `>` character.

The syntax for redirecting standard output is

```
command 1> file.txt
```

or

```
command > file.txt
```

If the descriptor is omitted standard output is assumed.

If the file does not exist it is created. If the file exists the action taken depends on the setting of the **noclobber** shell variable. When the **noclobber** option is turned on it prevents redirected output from destroying an existing file¹. In this case you will get an error message.

¹To set **noclobber** in bash use the command `set -o noclobber`.

If you wish to override the noclobber option you must use the redirection override operator `>|`.

The syntax for redirecting standard output with the noclobber override is

```
command >| file.txt
```

On the other hand, if you want to append the output to the file, the redirection operator is `>>`.

The syntax for redirecting standard output and appending to a file is

```
command >> file.txt
```

If the file does not exist UNIX will create it. If it exists the output from the command will be added to the end of the file. The noclobber option has no affect on the append redirection.

Redirecting Errors

By default the standard error stream is combined with the standard output stream and so they are displayed on the terminal together.

Example 1.4: If you try and list files that do not exist `ls` will output a message on standard error. Files that do exist are listed on standard out.

```
prompt: ls -l bashref.pdf make.pdf
ls: make.pdf: No such file or directory
-rw-r--r--  1 guest  guest  729678 Jan 29 14:35 bash.pdf
```

Standard error can be redirected to a file. Unlike standard output and standard input the stream descriptor must be used when redirecting standard error.

Example 1.5:

```
prompt: ls -l bashref.pdf make.pdf 2>ls.error
-rw-rw-r--  1 guest guest  729678 Jan 29 14:35 bash.pdf
prompt: cat ls.error
ls: make.pdf: No such file or directory
```

Example 1.6:

```
prompt: ls -l bashref.pdf make.pdf >ls.out 2>ls.error
prompt: cat ls.out
-rw-rw-r--  1 guest guest  729678 Jan 29 14:35 bash.pdf
prompt: cat ls.error
ls: make.pdf: No such file or directory
```

If you want both outputs to be written to the same file, it is not as simple as specifying the file name twice². To write output to the same file you must tell the shell that the second file is the same as the first by using the stream descriptor.

Example 1.7: Redirecting standard output and standard error to the same file

```
prompt: ls -l bashref.pdf make.pdf 1>ls.out 2>&1
prompt: cat ls.out
ls: make.pdf: No such file or directory
-rw-rw-r--  1 guest guest  729678 Jan 29 14:35 bash.pdf
```

²The reason is that when the second stream is to be redirected to the file it has already been opened for writing by the first stream.

The construct `2>&1` tells the shell to direct stream descriptor 2 to stream descriptor 1.

The order in which the redirection appears is important when redirecting two streams to the one file. You must first redirect one of the streams to a file—then the other stream to that descriptor. The reason is simple, the command `2>&1` copies the current descriptor for standard out to standard error and if it is executed first then the current descriptor for standard out points to the terminal.

Redirecting to the file `/dev/null`

UNIX has a special file called `null` found in the device directory `/dev`. Any output directed to this special device is gone for ever. If you do not wish to keep a copy of any output it should be redirected to `/dev/null`.

Example 1.8: Redirecting standard error to `/dev/null`

```
prompt: ls -l bashref.pdf make.pdf 2>/dev/null
-rw-rw-r-- 1 guest guest 729678 Jan 29 14:35 bash.pdf
```

1.2.3 Pipes

We often need to use a series of commands to complete a task. For example, the command `ps -aux` will list all processes on the system. If you want a hard copy of this list you first store the output in a file, then print the file using the `lpr` command.

Example 1.9: Print the output of the `ps` command

```
prompt: ps -aux > ps.txt
prompt: lpr ps.txt
prompt: rm ps.txt
```

We can avoid the creation of a temporary file by using a pipe. A pipe is an operator (the symbol '|') directing the shell to make the output of the lefthand command become the input of the righthand command.

Example 1.10: Print the output of the `ps` command using a pipe

```
prompt: ps -aux | lpr
```

Example 1.11: Convert a GIF image to a JPEG image

```
prompt: giftopnm < image.gif | pnmtojpeg > image.jpg
```

Example 1.12: Convert a GIF image to a JPEG image, shrinking it by a factor of 4 in the process.

```
prompt: giftopnm<image.gif | pnmscale 0.25 | pnmtojpeg>image.jpg
```

Exercise 1.13: Using the commands `ls` and `wc` how would you quickly find the total number of files in a directory? The number of GIF files?

Exercise 1.14: The `tee` command is a useful command. Explain why? Give examples of its use.

1.2.4 Command Lists

A single command can appear on the command line or a sequence of commands can be placed on one line. Multiple commands on one line are executed sequentially

Sequential List

To execute a sequence of commands each command is separated by a semicolon. There is no direct relationship between the commands. The commands are executed sequentially.

Example 1.15:

```
prompt: cd /etc/mail; ls *.cf
sendmail.cf  submit.cf
```

Example 1.16:

```
prompt: echo -e "\n      A Leap Year"; cal 2 2020
```

```
      A Leap Year
      February 2020
Su Mo Tu We Th Fr Sa
                1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

Conditional Lists

The semicolon creates a list of independent commands, the shell also recognises a conditional list—a list of dependent commands. A conditional list is a sequence of commands separated by the operators `&&` or `||`. The operators `&&` and `||` work like the C operators with the same symbols. As in C, the shell applies *short circuit evaluation* to the “and” (`&&`) and “or” (`||`) operators. Short circuit evaluation will not evaluate the second operand if evaluation of the first is sufficient to determine the result.

An “and” list has the form

```
command1 && command2
```

command2 is executed if, and only if, command1 returns an exit status of zero (true).

An “or” list has the form

```
command1 || command2
```

command2 is executed if, and only if, command1 returns a non-zero exit status (false).

Example 1.17: If the return value (exit status) of `diff` is false only then will the second command be executed.

```
prompt: diff file1 file2 >/dev/null || echo "files are different"
```

Note: In the UNIX shell the “exit status” or “return value” of a command is an integer. The standard return value for the successful completion of a command is 0, an error condition is a value other than zero. The reason is that there is normally only one success state but many error states. This means that for shell conditionals a return value of zero is *True* and a value other than zero is *False*. This is the reverse to C conditionals.

Exercise 1.18: What is the result of the following command

```
prompt: cp image.jpeg temp.jpeg && echo "Copy Successful"
```

If `image.jpeg` exists? If it does not exist?

Exercise 1.19: What is the result of the following command

```
prompt: cp image.jpeg temp.jpeg || echo "Copy Failed"
```

If `image.jpeg` exists? If it does not exist?

Exercise 1.20: What is the result of the following command

```
prompt: cd test && cat myfile.txt
```

if the directory does not exist? If the file does not exist?

Grouped Lists

A list of commands can be grouped so that they are executed as a unit. The syntax is

```
( command list )
```

Placing a list of commands between parentheses causes a subshell to be created, and each of the commands in the list are executed in that subshell. A subshell is a new shell process started by the current shell. The new shell is passed the list of commands to execute and the current shell waits for the subshell to terminate. One advantage of having a subshell execute a list of commands is that only the output of the subshell need to be redirected.

Example 1.21:

```
prompt: ( echo -e "\nList of JPEG images"; ls *.jpeg ) > list
prompt: cat list
```

```
List of JPEG images
image01.jpeg      image04.jpeg      image07.jpeg
image02.jpeg      image05.jpeg      image08.jpeg
image03.jpeg      image06.jpeg      image09.jpeg
```

1.2.5 Here Documents

A “here document” is a way of getting multiline text to a command from the command line.

Example 1.22:

```
prompt: cat >newfile <<EOT
This is the first line of the document.
Any typed text is automatically sent to
the cat command
EOT
prompt: cat newfile
This is the first line of the document.
Any typed text is automatically sent to
the cat command
```

The key to a “here document” is the << operator. This tells the shell to send all input from the keyboard to the command. The shell stops sending input to the command when the *exact* string to the right of the << is found on a line by itself. It must be the exact string and spaces are important—“ END” is different to “END”.

1.2.6 Command Substitution

If ``command`` appears on the command line or in a “here document” then the result of the command will replace it.

Example 1.23:

```
prompt: echo There are `who | wc -l` logins
There are 5 logins
```

Example 1.24:

```
prompt: cat >newfile <<EOT
Today's date is `date`,
there are `ps -aux | wc -l` processes running
EOT
prompt: cat newfile
Today's date is Thu Mar  6 13:33:44 EST 2003
there are      102 processes running
```

In Bash the construct `$(command-string)` is a synonym for ``command-string``.

1.2.7 Job Control

The shell's definition of a “job” is a command or set of commands entered on one command line. As UNIX is a multitasking operating system, we can run more than one job at a time. However, when we start a job in the foreground, the standard input and output are locked. They are available exclusively to the current job until it completes. To allow multiple jobs access to standard input and standard output, UNIX defines two types of jobs: foreground and background.

Foreground Jobs

A *foreground job* is any job run under the active supervision of the user. While it is running, no other jobs may be started. But it may be killed by the user or suspended. To kill or terminate a running job it must be sent the **KILL** signal. From the keyboard the KILL signal can be sent to the foreground job by pressing `<ctrl>c` (that is the hold down the control key then press `c`).

A foreground job can be suspended, so other commands can be run by the user, then restarted. To suspend a job press `<ctrl>z`, to restart the suspended job type `fg` on the command line.

Example 1.25: Suspending a process and then restarting it

```
prompt: info make
<ctrl>z
[1]+  Stopped                  info make
prompt: date
Thu Mar  6 17:12:36 EST 2003
prompt: fg
```

Background Jobs

Jobs run in the background free the shell to resume processing commands. Note that foreground and background jobs share the output. Any messages sent to standard output by the background will be mingled with the messages sent from the foreground job. It is normally a good idea to redirect standard output of background jobs.

To start a job running in the background place an ampersand & as the last character on the command line.

Example 1.26: Starting a job in the background:

```
prompt: emacs file.txt &
[3] 1874
prompt:
```

In the example above the number in square brackets is the job number, the other number is the process number.

Example 1.27: Push an already running foreground job into the background:

```
prompt: emacs file.txt
<CTRL>Z
[2]+  Stopped                  emacs file.txt
prompt: bg
[2]+ file.txt &
prompt:
```

First suspend it then push it into the background with `bg` where it will resume running.

1.2.8 Variables

Variables are created by the shell when they are assigned a value.

Example 1.28: When a variable is assigned a value it is created:

```
prompt: str='Hello World'
prompt: echo $str
Hello World
```

When assigning a value to a variable use the assignment operator =. There must not be spaces around the = operator.

To access the value of a variable, the name of the variable must be preceded by a dollar sign as shown in example 1.28 above.

Example 1.29: Using variables:

```
prompt: me='whoami'
prompt: echo My login name is $me!
My login name is guest!
```

Example 1.30: Using variables:

```
prompt: dir=~ /report
prompt: cd $dir
prompt: pwd
/home/guest/report
```

Example 1.31: Using variables:

```
prompt: suffix=txt
prompt: prefix=name
prompt: file=${prefix}.${suffix}
prompt: echo $file
name.txt
```

For clarity braces may be put around the variable name when referencing it. The braces are required when the character following the variable name is not to be interpreted as part of the name.

Environment Variables

When variables are created they are by default local to the shell in which they were created. Any program started in the shell (called a *child process* of the shell) cannot access the local variables. Variables that can be accessed by a child process of the shell are called *Environment Variables*.

Environment variables are used to communicate to processes the environment that they are running in. A user can use environment variables to modify the behaviour of processes.

Example 1.32: The command `printenv` will list all the currently defined environment variables:

```
prompt: printenv
HOST=D113-00
TERM=xterm
SHELL=/bin/bash
HISTSIZE=500
HOSTDISPLAY=D113-00.sci.usq.edu.au:0.0
LC_ALL=POSIX
PAGER=/usr/bin/less
EDITOR=xemacs
DISPLAY=:0.0
SHLVL=2
HOME=/home/guest
...
...
```

When you login many environment variables are created for you by the system.

Example 1.33: An example of how a C program could access environment variables:

```
1 | /* A simple C program to print Environment Variables */
2 | #include <stdlib.h>
3 | #include <stdio.h>
4 |
5 | int main(int argc, char *argv[], char *env[])
6 | {
7 |     int i=0;
8 |
9 |     while ( env[i] ) {
10 |         printf("%s\n",env[i]);
11 |         i++;
```

```
12 | }  
13 | }
```

Exercise 1.34: Compile the code of example 1.33 and compare the output from the code and the output from the `printenv` command.

To turn a local variable into an *environment variable* the shell has to be told to *export* the variable.

Example 1.35: Exporting variables:

```
prompt: file=project.txt  
prompt: export file  
prompt: printenv | grep file  
file=project.txt
```

The two step process of assigning a value to a variable and then exporting the variable can be done in one step:

```
prompt: export file=project.txt  
prompt: printenv | grep file  
file=project.txt
```

Exercise 1.36: Create your own exported variables. Use the `printenv` command and the program of example 1.33 to show that your variables are now exported and available to child processes.

When you login the system predefines a default set of environment variables that define a standard user environment. Some examples are:

HOME The path of the users home directory.

SHELL The name of the running shell.

TERM The type of terminal you are using.

PATH The search path used by the shell to find a command executable.

EDITOR The editor preferred by the user.

PWD The current working directory

Variable Expansion

Bash allows for variable or parameter expansion. The standard for accessing a variable's value is `$name` or `${name}`. The latter form is used to protect a variable name or can be used to perform variable expansion.

Variable expansion is where a variable's value is modified before it is substituted. For example, a variable that contains a path to a file might have the leading directories removed before the value is substituted.

Example 1.37: Substitute a default value for an undefined variable. The variable name is undefined:

```
prompt: echo $name                # undefined variable

prompt: echo ${name:-fred}
fred
prompt: name=george
prompt: echo ${name:-fred}
george
```

When name is undefined the default value of fred is substituted. When name is defined then its value is substituted.

There are a number of parameter expansions. A few examples are

\${param:=word} — if param is undefined then assign it the value word and substitute that value.

\${param#pattern} — remove the leading pattern. Where pattern can use the file matching wildcards.

\${param%pattern} — remove the trailing pattern. Where pattern can use the file matching wildcards

See the Bash manual for the complete list of variable or parameter substitution.

Example 1.38: Find the name of the current working directory:

```
prompt: echo $PWD
/home/users/guest/projects
prompt: echo ${PWD##*/}
projects
```

Example 1.39: Find the path to the current working directory

```
prompt: echo $PWD
/home/users/guest/projects
prompt: echo ${PWD%/*}
/home/users/guest
```

1.2.9 Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent variable expansion.

We have seen above that there are various forms of shell expansion. That is, various substitutions are performed — command substitutions, variable substitution, filename expansion etc. How and when these expansions are performed can be controlled by the judicious use of quotes.

Quotes are also used to group words — the special meaning of the space character as a command or parameter separator is disabled.

Escape Character

The backslash is the Bash escape character — it preserves the literal value of the next character that follows.

Example 1.40:

```
prompt: echo \$HOME contains the path to the users home
$HOME contains the path to the users home
```

The `$` becomes a simple character.

If the backslash is immediately followed by the newline character then the newline is treated as a line continuation.

Single Quotes

Enclosing characters within single quotes (`'`) preserves the literal value of each character within the quotes. No substitutions of any kind are performed.

Example 1.41:

```
prompt: echo 'The current path is $PWD'
The current path is $PWD
```

The single quotes suppresses the variable substitution.

Double Quotes

Enclosing characters in double quotes (") preserves the literal value of all characters within the quotes, with the exception of \$, ', and n.

This means that variable substitution is performed within double quotes.

Example 1.42:

```
prompt: echo "The current path is \"${PWD}\""  
The current path is "/home/users/guest"
```

The double quotes does not suppress the variable substitution. To include literal double quotes within the string they must be escaped

1.3 Shell Programming

To date in this module we have discussed the basic shell features. These are the features most often used when interacting with the shell directly. Interactive sessions work well for simple commands but when problems become larger and more complex, especially if they need to be solved repetitively, then commands need to be saved in a file.

1.3.1 Shell Scripts

A shell script is a text file that contains shell commands. The shell reads the script file and executes the commands specified on each line in turn.

Example 1.43: The following is the contents of the file `script1`

```
1 | #!/bin/bash
2 | # A simple script to print out a string
3 |
4 | echo 'Hello World'
```

To run this script:

```
prompt: script1
Hello World
```

Example 1.43 shows the source for a basic shell script. Some important things to note about this script:

- The first line of a script should be the *Interpreter Designator* line. All UNIX shells recognise this line. This line specifies which process needs to be started to interpret the script.

The designator line begins with a `#!`, followed by the path to the interpreter. If this line is missing the users current shell will be used as the interpreter and this might not be correct.

Incorporate the *Interpreter Designator* line in all scripts to ensure the correct interpreter is used.

- The second line is a comment line. All characters following a `#` are ignored. Blank lines are also ignored.
- To be able to run the script the file `script1` must be made executable. That is,

```
chmod u+x script1  
or  
chmod 0700 script1
```

1.3.2 Special Variables

The shell defines a number of variables or parameters that are useful in shell scripts. These parameters cannot be assigned values.

0 This variable contains the name of the script.

\$ This variable contains the process id of the script.

This variable contains the number of arguments the script was called with.

* This variable contains the arguments as a string.

1..9 These are positional parameters. The variable 1 contains the first argument, 2 contains the second argument, etc. There are at most only nine positional parameters.

As with any shell variable to access the contents of the special readonly variables just place a \$ in front of the variable.

Example 1.44: The following script shows some of the shell special variables

```
1 | #!/bin/bash  
2 | # A simple script showing some of the special variables  
3 |  
4 | echo "The full name of this script is $0"
```

```
5 |
6 | name=${0##*/}
7 |
8 | echo "The name of this script is $name"
9 |
10 | echo "The process ID of this script is $$"
11 |
12 | echo "This script was called with $# parameters"
13 |
14 | echo "The parameters are \"$*\\""
15 |
16 | echo "Parameter 1 is - $1"
17 | echo "Parameter 2 is - $2"
18 | echo "Parameter 3 is - $3"
```

```
prompt: code/script2 fred george allison
The full name of this script is code/script2
The name of this script is script2
The process ID of this script is 3501
This script was called with 3 parameters
The parameters are "fred george allison"
Parameter 1 is - fred
Parameter 2 is - george
Parameter 3 is - allison
```

1.3.3 Arithmetic Expressions

The Bash shell has the ability to perform integer arithmetic. There are three ways to incorporate arithmetic expressions into a script: using the `let` builtin command,

or the operators `$(expression)` or equivalently `$((expression))`.

The shell will not check for integer overflow but will flag a divide by zero error.

The arithmetic operators the shell understands are all the C integer operators. Precedence and associativity are the same as in the C language.

Example 1.45: Examples of arithmetic expressions:

```
prompt: let y=356
prompt: let x=y*235+108
prompt: echo $x
83768
prompt: echo $((y * x))
29821408
prompt: echo ${y%2}
0
prompt: (( y = 23 ))      # C-like Constructs
prompt: (( y++ ))
prompt: echo $y
24
prompt: (( x=y>23?0:1 ))
prompt: echo $x
0
```

Some things to note from the example 1.45 above:

- No spaces are allowed in the `let` expressions.
- The `$variable` syntax is not allowed in `let` expressions.
- Spaces are allowed in `$(expression)` or `$((expression))`

- The `$variable` syntax is optional in `$(expression)` or `$((expression))`
- The construct `((expression))` allows C-like manipulation of variables.

Any variable can be used in an arithmetic expression whether it contains an integer or not. If the variable does not contain an integer the shell treats it as zero.

Example 1.46: An example of using a variable that is not an integer in an arithmetic expression

```
prompt: count=fred
prompt: echo $count $((count+5))
fred 5
```

1.3.4 Conditional Expressions

Conditional expressions return a *true* or *false* result. Conditional expressions are used in control structures such as `if`, `while`, `until`, etc. (see §1.3.5).

In UNIX the convention is that zero is *true* and anything other than zero is *false*. This is the reverse of the C convention. The reason for it is straight forward — the convention in UNIX is that programs should terminate with an integer exit status. Programs normally have only one success status, a return value of zero, but can have many error results.

Any UNIX program that returns an integer status code (which should be all of them) can be used in a conditional expression.

Example 1.47: Examples of the return status of UNIX programs

```
prompt: cd xxx >/dev/null 2>&1
prompt: echo $?
1
```

The special parameter `?` contains the return status of the last command. In this case the directory does not exist and the return status is false.

```
prompt: cd . >/dev/null 2>&1
prompt: echo $?
0
```

The current directory always exists—so the change directory command was successful.

It is not strictly true that the current directory always exists. Outline a situation where the current directory would not exist!

In fact any program or shell expression that returns an integer status code can be used, for example the `let` or `((expression))` construct return an exit status of 0 if the arithmetic expressions they evaluate expand to a non zero value.

The Test Command

The `test` command is a program that was used by the Bourne shell to perform logical comparisons of strings, integers and test the status of files. The `test` command returns 0 or 1, true or false.

Exercise 1.48: Read the man page for the `test` command.

The old `test` command can be used in shell scripts—but `bash` has introduced the equivalent construct `[expression]` which is a synonym of the `test` command. The one advantage of the `[expression]` construct is that it is built in to the shell (Note: there must be a space at either end of `expression`).

Example 1.49: Some examples of the `test` command:

```
prompt: test 3 -lt 2; echo $?
1                               # false
prompt: name=fred
prompt: test -x $name; echo $?
1                               # false
prompt: ls -l file1 file2 dir1
drwxr-xr-x  2 guest  guest    4096 Mar 18 12:16 dir1
-rw-r--r--  1 guest  guest         0 Mar 18 12:16 file1
-rw-r--r--  1 guest  guest    19 Mar 18 12:16 file2
prompt: test -f file1; echo $?
0                               # true
prompt: test -f dir1; echo $?
1                               # false
prompt: [ -s file1 ]; echo $?
1                               # false
prompt: test -s file2; echo $?
0                               # true
prompt: test -s file2 -a -d dir1; echo $?
0                               # true
prompt: [ -e dir1 -o -s file1 ]; echo $?
0                               # true
```

`Bash` has also introduced the `[[expression]]` found in the `Korn` shell. This is the *extended test command* that has a syntax similar to conditional expressions from

other languages—that is the logical operators `!`, `&&` and `||` are recognised.

1.3.5 Control Structures

Control structures are required if a language is to be useful as a programming language. The shell has the standard control structures found in any programming language—branching structures `if` and `case`, and looping structures `for`, `while`, `until` etc.

if-then-else

The syntax of the basic `if`-structure is:

```
if test-command-list
then
    command-list
fi
```

Normally one key word is placed per line but a more compact form can be constructed using a semicolon to terminate the command lists:

```
if test-command-list; then command-list; fi
```

Example 1.50: Example of a simple `if`-structure in a script:

```
1 | #!/bin/bash
2 | # A simple script using an if-statement
3 |
```

```
4 | name=${0##*/}
5 |
6 | if [ $# -ne 1 ]; then
7 |     echo "Usage: $name parameter"
8 |     exit 1
9 | fi
10 |
11 | echo "The command line parameter is '$1'"
```

```
prompt: script3
Usage: script3 parameter
prompt: script3 fred
The command line parameter is 'fred'
```

The general form of the if-structure is:

```
if test-command-list; then
    command-list
elif another-test-command-list; then
    another-command-list
    ...
    ...
else
    alternate-command-list
fi
```

Any number of the `elif` (*else if*) lines can appear but only one `else` line can be present.

Example 1.51: Example of a if-elif-else-structure in a script:

```
1  #!/bin/bash
2  # A simple script using an if-elif-else statement
3  # Turn a score into a grade
4
5  name=${0##*/}
6
7  if test $# -ne 1; then
8      echo "Usage: $name score"
9      exit 1
10 fi
11
12 if [ $1 -lt 0 -o $1 -gt 100 ]; then
13     echo "The score must be in the range 0-100"
14     exit 2
15 fi
16
17 score=$1
18
19 if ((score >= 85)); then
20     grade='HD'
21 elif ((score >= 75)); then
22     grade='A'
23 elif ((score >= 65)); then
24     grade='B'
25 elif ((score >= 50)); then
26     grade='C'
27 else
28     grade='F'
29 fi
30
31 echo "The grade is $grade"
```

```
prompt: script4 72
The grade is B
prompt: script4 105
The score must be in the range 0-100
```

Exercise 1.52: Explain the three different types of tests used in the three `if` statements of example 1.51.

Exercise 1.53: Add comments to thoroughly document the script of example 1.51

case

The Bash shell implements a version of the `case` statement. Given a string and a list of *pattern* alternatives, the `case` statement matches the string against each of the patterns in sequence. The first pattern that matches the string gets the action.

The syntax of a `case` statement is:

```
case word in
  pattern1) command-list;;
  pattern2) another-command-list;;
  pattern3|pattern4) another-command-list;;
  ...
  ...
esac
```

Things to note about the case syntax above:

- A case statement is ended by the `esac` keyword—which is case spelt backwards.
- A pattern is delimited by a `)` character.
- A pattern is the standard file globbing wildcard pattern with the addition of the `'|'` (or) operator.
- The command list is terminated with `;;`
- The shell case statement is not like its C equivalent—it does not drop through. When the *command-list* has been executed (if any matches were found) the shell jumps to the first command after the `esac` statement.

Example 1.54: An example of a simple case statement:

```
1  #!/bin/bash
2  # A simple script using a case statement
3
4
5  read -p "Enter a single digit> " digit
6
7  case $digit in
8      0) word='Zero';;
9      1) word='One';;
10     2) word='Two';;
11     3) word='Three';;
12     4) word='Four';;
13     5) word='Five';;
14     6) word='Six';;
15     7) word='Seven';;
```

```
16 | 8) word='Eight';;
17 | 9) word='Nine';;
18 | *) word='Not a Single Digit';;
19 | esac
20 |
21 | echo "You entered a '$word'"
```

```
prompt: script5
Enter a single digit> 8
You entered a 'Eight'
prompt: script5
Enter a digit> 0
You entered a Zero
prompt: script5
Enter a digit> x
You entered a 'Not a Single Digit'
```

Exercise 1.55: The script of example 1.54 uses the shell builtin `read`. Read the bash man page on the builtin command `read`.

What could the `read -s` command be used for in a shell script?

What is the effect of `read -t 10` ?

Example 1.56: An example of a more complex case statement:

```
1  #!/bin/bash
2  # A script using a case statement
3  # A greeting script
4
5  hour=`date +%H:%M`
6
7  case $hour in
8  0?:??|1[01]:??) echo "Good Morning. It's $hour A.M.>";;
9      12:??) echo "Good Afternoon. It's $hour P.M.>";;
10     1[3-7]:??) pm=$(( ${hour%:*} - 12 )
11                min=${hour#*:}
12                echo "Good Afternoon. It's $pm:$min P.M.>";;
13  1[89]:??|2?:??) pm=$(( ${hour%:*} - 12 )
14                min=${hour#*:}
15                echo "Good Evening. It's $pm:$min P.M.>";;
16     *) echo "Sorry, I don't know the time"
17  esac
```

```
prompt: script6
Good Afternoon. It's 5:15 P.M.
```

Exercise 1.57: Add comments to thoroughly document the script of example 1.56

Exercise 1.58: The script of example 1.51 is expecting one parameter with a value between 0 and 100.

What happens if the parameter is not an integer? Why is the `if`-statement failing?

Replace the `if`-statement that is failing when the parameter is not an integer with a `case`-statement to detect when the parameter is not an integer between 0 and 100.

Exercise 1.59: Repeat exercise 1.58 but this time use an `egrep` command as the conditional expression in an `if`-statement to check that the command line parameter is an integer.

for

The shell `for`-loop can be considered a *list controlled* loop. That is, the number of elements in the control list controls the number of iterations of the loop. If there are five elements in the loop, the body of the loop will be executed five times.

The basic syntax of a `for`-loop is:

```
for arg in list
do
    command-list
done
```

During each pass through the loop, `arg` takes on the value of each successive word in the list.

Example 1.60: An example of a basic for-loop:

```
for day in Sunday Monday Tuesday Wednesday \  
        Thursday Friday Saturday  
do  
    echo $day  
done
```

or alternately

```
dow=' Sunday  
    Monday  
    Tuesday  
    Wednesday  
    Thursday  
    Friday  
    Saturday'  
  
for day in $dow  
do  
    echo $day  
done
```

The shell for-loop recognises a couple of list constructing shortcuts—filename globbing and command line arguments.

If the list in a for-loop contains filename globbing the wildcard string will be expanded to a list of filenames in the current directory.

Example 1.61: An example of using filename globbing in a for-loop

```
1  #!/bin/bash
2  # A script using a for-loop and
3  # filename globbing
4  #
5  # List files and types
6
7  for file in *
8  do
9      if [ -h $file ]; then
10         echo "$file (soft link)"
11     elif [ -x $file ]; then
12         echo "$file (executable)"
13     elif [ -d $file ]; then
14         echo "$file (directory)"
15     elif [ -p $file ]; then
16         echo "$file (named pipe)"
17     elif [ -S $file ]; then
18         echo "$file (socket)"
19     else
20         echo $file
21     fi
22 done
23
24 echo
```

Exercise 1.62: Why, in example 1.61 does the test for a symbolic link come first? What happens if it does not?

Exercise 1.63: The same effect of the shell filename globbing in example 1.61 could be achieved with command substitution using the `ls` command. Rewrite example 1.61 using command substitution.

If the “in *list*” part of a `for`-loop is missing then the `for`-loop assumes the list is “\$@”, the command-line argument list.

Example 1.64: An example of using a `for`-loop without the “in *list*”.

```
1 | #!/bin/bash
2 | # A script using a for-loop and
3 | # the command-line argument list
4 | #
5 |
6 | for arg
7 | do
8 |     echo \'$arg\'
9 | done
10 |
```

Exercise 1.65: What is the difference between the special shell variables `$*` and `$@`

Write a shell script to test the following `for`-loops

```
for arg in $*;
do
...

```

```
done

for arg in $*;
do
    ...
done

for arg in "$@";
do
    ...
done

for arg in "$*";
do
    ...
done
```

What is stored in `arg` each time through each loop? How many times is each `for`-loop body executed?

Exercise 1.66: Comment the following script—line by line:

```
1 | #!/bin/bash
2 | # A script to list files greater than 100k
3 | # in a directory
4 | #
5 |
6 | size='100k'
7 | directory=${1-'pwd'}
8 |
```

```
9 | echo "Files greater then $size in \"$directory\""  
10 |  
11 | for file in "$( find $directory -size +$size )"  
12 | do  
13 |     echo "$file"  
14 | done | sort
```

Hints:

- What happens if the script is called with a directory path on the command line? What is the value of `directory`?
 - What happens if the script is called without any command line parameters? What is the value of `directory`?
 - What happens if the `| sort` is removed?
-

while

The `while`-loop is a basic *command-controlled* loop. The exit status of the loop command is tested at the start of the loop, the loop body is executed while the exit status of the loop command is true (zero). When the exit status of the loop-command becomes false (nonzero) the loop is terminated.

The basic syntax of a `while`-loop is:

```
while test-commands  
do  
    command-list  
done
```

The return status of the while-loop is the return status of the last command executed in the *command-list*.

Example 1.67: An example of using a while-loop:

```
1 | #!/bin/bash
2 | # A script demonstrating a while loop
3 | #
4 | # Add the numbers entered
5 |
6 | cat<<EOT
7 | This script adds numbers entered from
8 | the keyboard. When all the numbers have
9 | been entered hit ^d (control-d: eof) to
10 | see the total
11 |
12 | EOT
13 |
14 | sum=0
15 |
16 | while read -p "Enter a number : " data
17 | do
18 |     (( sum = sum + data ))
19 | done
20 |
21 | echo -e "\n          Total is : $sum"
```

Exercise 1.68: Add comments to the script of example 1.67—line by line.

Example 1.69: An example of using a while-loop to process the command-line arguments of a script:

```
1 | #!/bin/bash
2 | # A script demonstrating a while loop
3 | #
4 | # Grab the command line arguments
5 |
6 | name=${0##*/}
7 |
8 | if [ $# -eq 0 ]; then
9 |     echo "Usage: $name arg1 [ [arg2] [arg3] ... ]"
10 |     exit 1
11 | fi
12 |
13 | count=1
14 |
15 | while [ $# -gt 0 ]; do
16 |     echo "Arg[$count] is '$1'"
17 |     shift
18 |     (( count++ ))
19 | done
```

Exercise 1.70: Add comments to the script of example 1.69—line by line, explaining in detail the effect of the shell inbuilt command `shift`.

until

The `until`-loop has the same structure as the `while`-loop. The difference is the loop body is executed while the exit status of the loop command is false (nonzero). When the exit status of the loop-command becomes true (zero) the loop is terminated. This is the reverse of the `while`-loop.

The basic syntax of an `until`-loop is:

```
until test-commands
do
    command-list
done
```

Example 1.71: An example of using an `until`-loop to process the command-line arguments of a script:

```
1 | #!/bin/bash
2 | # A script demonstrating an until loop
3 | # and command-line processing
4 | #
5 | # List the regular files of a directory
6 | # greater than a given size
7 |
8 | name=${0##*/}
9 |
10 | Usage="Usage: $name [-h] [-s N] [directory]"
11 |
12 |
13 | until [ $# -eq 0 ]
14 | do
```

```
15     case $1 in
16         -s)  shift
17             size=$1
18             shift;;
19         -h)  echo $Usage; exit 0;;
20         *)  directory=$1;;
21     esac
22 done
23
24 directory=${directory:-`pwd`}
25
26 if [ ! -d $directory ]; then
27     echo "'$directory' is not a directory"
28     exit 1
29 fi
30
31
32 arg=''
33
34 if [ $size != '' ]; then
35     arg="-size +$size"
36 fi
37
38 find $directory $arg -type f -exec ls -l {} ';'

```

Exercise 1.72: Add comments to the script of example 1.71—line by line.

Exercise 1.73: Example 1.71 does not have a very helpful help message. Using a *here*-document improve the message returned when the `-h` command line flag is set.

Exercise 1.74: One looping command that has not been discussed is the `select`-loop. Read the `select-loop` section of the Bash man page. Write a test script to demonstrate the use of the `select`-loop

Loop Control

The shell has two builtin commands that can interrupt the flow of a `for`, `while`, `until` or `select` loop.

break The `break` builtin command will exit from the innermost loop. If the command is of the form

```
break n
```

then the n^{th} enclosing loop is exited.

continue Start the next iteration of an enclosing loop. if the command is of the form

```
continue n
```

then the next iteration of the n^{th} enclosing loop is started.

1.3.6 Functions

Shell functions are similar to functions in other languages. Functions are a way of grouping commands together for *later* execution using a single name. The list of commands associated with the function are executed when the function name is used as a simple command name. Functions can be recursive, and no limit is placed on the number of recursive calls.

The syntax for defining a function called *fname* is:

```
function fname
{
    command-list ;
}
```

or

```
fname ()
{
    command-list ;
}
```

A few things to note about shell functions::

- the reserve word `function` is optional. If it is present then the parenthesis `()` are optional.
- the body of the function is the *command-list* between `{` and `}`. This list is executed whenever *fname* is specified as the name of a command.
- the *command-list* must be terminated with a semi-colon or a newline

- the curly braces that surround the body of the function must be separated from the body by blanks or newlines.
- when a function is executed, the arguments to the function become the positional parameters during its execution (positional parameter \$0 is unchanged).
- the variable FUNCNAME is set to the name of the function while the function is executing.
- if the builtin command `return` is executed in a function, the function exits and execution resumes with the next command after the function call.
- the exit status of the function is the exit status of the last command executed. If the `return` command has an optional numeric argument that is the functions return status.

Example 1.75: An example of using functions in a shell script:

```
1  #!/bin/bash
2  # A script demonstrating using functions
3
4  name=${0##*/}
5
6  Usage="Usage: $name parameter"
7
8
9  # a function that returns true if the
10 # parameter is an integer
11 # (an example from the Bash distribution
12 # with errors corrected )
13
14 isint()
15 {
```

```
16     case "$1" in
17         [-+] )    return 1;;
18     [-+]*[^0-9]*) return 1;;
19         [-+]* )   return 0;;
20     *[^0-9]*)    return 1;;
21         *)        return 0;;
22     esac
23 }
24
25 # a help function
26 fhhelp ()
27 {
28     cat<<EOT
29 $Usage
30     This script is an example of using functions.
31     It will tell you if the parameter is an integer
32     or not.
33
34     The 'isint' function (with many others) can be
35     found in the examples of the Bash distribution.
36 EOT
37
38     exit 1
39 }
40
41 if [ $# != 1 ]; then
42     fhhelp
43 fi
44
45 if isint $1; then
46     echo '... yep, it was an integer'
47 else
48     echo '... nope, not an integer'
```

```
49 | fi
```

Exercise 1.76: Add comments to the script of example 1.75—line by line.

Exercise 1.77: The help function in example 1.75 is called `fhelP`. Why can't a function be called `help`?

1.4 Script Examples and Exercises

Ex. 1.78: Write a shell script to mimic the behaviour of the `cat` command.

(Hint: Use the `read` builtin with the `-r` flag.)

Ex. 1.79: Write a shell script to convert all the filenames that appear on the command line into lowercase. For example:

```
prompt> ls
File02.txt    file03.txt
File05.txt    file04.txt
File07.txt    file06.txt
file01.txt    file08.txt
prompt: lowercase file*
prompt> ls
```

```
file01.txt    file05.txt
file02.txt    file06.txt
file03.txt    file07.txt
file04.txt    file08.txt
```

Ex. 1.80: Write a simple rename script. That is, a script that will rename a set of files. It should be called in the following manner

```
prompt: rename pattern1 pattern2
```

All files that match `pattern1` should be renamed using `pattern2`. Use the two patterns with the `sed` utility to change the file's name.

Ex. 1.81:

```
1  #!/bin/bash
2  # Search for the existence of a file in
3  # the directories listed in the user's PATH
4  # (Based on a script from the
5  # Bash distribution)
6
7  function inpath
8  {
9  pathlist=`echo $PATH | sed 's/^:/:./
10                                     s/::/:./g
11                                     s/:$/:./
12                                     s:/ /g`
13
14  for dir in $pathlist
15  do
16          [ -f $dir/$1 ] && return 0
17  done
18
```

```

19 | return 1
20 | }
21 |
22 | name=${0##*/}
23 |
24 | if [ $# -lt 1 ]; then
25 |     echo "Usage: $name filename [filename ...]"
26 |     exit 1
27 | fi
28 |
29 | while [ $# -ne 0 ] ; do
30 |     if inpath $1; then
31 |         echo "$1 - found"
32 |     else
33 |         echo "$1 - not found"
34 |     fi
35 |     shift
36 | done

```

Add comments to the script—line by line

Ex. 1.82: Modify the script of exercise 1.81 so that extra paths can be added to the end of the `pathlist` and therefore searched with the following parameters

```
script -p ./mnt/floppy:/mnt/zip/mydir mozilla netscape
```

Use the idea of a help-function to extend the usage message of the modified script.

Ex. 1.83:

```

1 | #!/bin/bash
2 | # A script to parse the password file
3 |

```

```
4 | name=${0##*/}
5 |
6 | if [ $# -ne 1 ]; then
7 |     echo "Usage: $name username"
8 |     exit 1
9 | fi
10 |
11 | IFS=:
12 |
13 | while read id fl uid gid user home shell
14 | do
15 |     if [ $1 == $id ]; then
16 |         echo "$id is user '$user'"
17 |         exit 0
18 |     fi
19 | done </etc/passwd
20 |
21 | echo "$1 was not found"
22 |
23 | exit 1
```

What does the Bash variable `IFS` represent?

Ex. 1.84: The following script uses the builtin function `getopts`

```
1 | #!/bin/bash
2 | # A script using the builtin 'getopts'
3 |
4 | name=${0##*/}
5 |
6 | shelp()
7 | {
8 |     echo "$name: an example of the 'optargs' builtin command
9 |
```

```
10  $Usage
11
12  $name does nothing except show how to use the bash
13  builtin optargs command. This builtin is very usefull
14  when your script needs a large number of command line
15  parameters. More importantly it allows command line
16  flags to be grouped which is easier for the user.
17  Options:
18     -h prints this help.
19     -d sets the dflag to true
20     -x sets the xflag to true
21     -f sets the xflag to true
22     -i N
23         sets the iflag to the value N
24     -s string
25         sets the sflag to the value of string"
26 }
27
28 Usage="Usage: $name [-hdxfl] [-i N] [-s N] file ..."
29
30 while getopts :hdxfls: opt; do
31     case $opt in
32         h) shelp;          exit 0;;
33         d) dflag=1;        echo "dflag=1";;
34         x) xflag=1;        echo "xflag=1" ;;
35         i) iflag=$OPTARG;  echo "iflag=$iflag" ;;
36         s) sflag=$OPTARG;  echo "sflag=$sflag";;
37         ?) echo "$name: $OPTARG: bad option."
38             echo "Use -h for help."
39             exit 2;;
40     esac
41 done
42
```

```

43 | shift $((OPTIND - 1))
44 |
45 | if [ $# -lt 1 ]; then
46 |     echo -e "$Usage\nUse -h for help."
47 |     exit
48 | fi
49 |
50 | while [ $# -ne 0 ]
51 | do
52 |     echo "Filename: $1"
53 |     shift
54 | done

```

Using this script what is the result of the following commands?

- (a) `script -h`
- (b) `script -dx file1 file2`
- (c) `script -d -x -i 100 file1 file2`
- (d) `script -dxi 100 file1 file2`
- (e) `script -s string file1`
- (f) `script -dxis 100 file1 file2`
- (g) `script -dx file1 -s string`

What are the advantages of the `getopt` builtin? What are its disadvantages?

Ex. 1.85: The following script uses the builtin function `eval`. It is a script to execute a command a given number of times

```

1 | #!/bin/bash
2 | # A script to repeat a command
3 | # (Based on a script from the

```

```
4 # bash distribution)
5
6 name=${0##*/}
7
8 Usage="Usage: $name [-p n] <count> <command>"
9
10 pause=''
11
12 if [ "$1" == '-p' ]; then
13     pause=$2
14     shift 2
15
16     let x=pause
17     if [ $x != $pause ]; then
18         echo "$Usage"
19         exit 1
20     fi
21 fi
22
23 count=$1
24
25 if [ $# -le 1 ] || [ -z "$count" ]; then
26     echo "$Usage"
27     exit 2
28 fi
29
30 shift
31
32 st=0
33 while [ $count -gt 0 ]; do
34     eval "$@"
35     st=$?
36     count=$((count - 1))
```

```
37 |     if [ ! -z "$pause" ]; then
38 |         sleep $pause
39 |     fi
40 | done
41 |
42 | exit $st
```

Run this script with the following command line input:

- (a) `script`
- (b) `script 2 ls`
- (c) `script -p 10 2 ls -l`

Read the bash man page on the builtin commands `eval` and `sleep`

Experiment with this script and add comments to every line.

There is a minor error in the script when the pause flag is used. After the command is executed for the last time the script will sleep before exiting—modify the script so that the last time through the loop the script will not sleep irrespective of the value of the `pause` variable.

Ex. 1.86: The bash shell recognises arrays. Read the bash man page section on arrays.

Write a script to output a random card from a deck of cards. For example:

```
prompt: card
3 of Diamonds
prompt: card
Jack of Clubs
```

(Hint: Store all 52 strings in an array and then use the construct `$((RANDOM%52))` to select card.)

1.5 Further Reading and References

- A description of the POSIX Globbing standard can be found on the man page `glob(7)`.
- A description of the POSIX Regexp standard can be found on the man page `regex(7)`.
- The Bash man page in PDF form can be found on the course web site.
- The “The Bash Beginner’s Guide” from *The Linux Document Project* (<http://www.tldp.org>) can be found on the course web site or DVD.
- The “Bash Programming Introduction” from *The Linux Document Project* (<http://www.tldp.org/>) can be found on the course web site or DVD.
- The Advanced Bash-Scripting Guide from *The Linux Document Project* (<http://www.tldp.org>) can be found on the course web site and the DVD.
- Many example scripts are distributed with the Bash source distribution. Under Linux the scripts can be found in `/usr/share/doc/bash-*`
- Many examples of system scripts can be found in the directory `/etc`. Especially in `/etc/init.d`
- Any book on UNIX that incorporates a chapter on shell scripting.