Writing Shell Scripts in UNIX

Combining sets of commands into one file, which then may be run to execute the other commands. This is extremely useful for backups, moving files, general housekeeping. A shell script is a file containing commands that could also be typed at the command line. When given execute permissions and called, the file is interpreted by the system and the commands are executed.

A simple Korn-shell script

To make sure that the correct shell is run, the first line of the script should always indicate which one is required. Below is illustrated a simple Korn-shell script: it just outputs the message "hello world":

```
#!/bin/ksh
echo "hello world"
```

Execute the script and observe the output

Comments and Commands

implies a comment in every line except the first line in the file. Here, the comment is used to indicate which shell should be used (ksh is the Korn Shell). Good practice to use comments to indicate what the script does.

As a bare minimum, the script must always contain the name of the author, the date written, a brief description of what the script does and some indication as to whther it works. The script should also contain comments showing the revision history.

Lines not preceded by a hash '#' character are taken to be UNIX commands and are executed. Any UNIX command can be used. For example, the script below displays the current directory name using the **pwd** command, and then lists the directory contents using the **ls** command.

```
#!/bin/ksh
# Script Written by Phil Irving and Andrew Hunter 18/6/98
# purpose: print out current directory name and contents
pwd
ls
```

Shell Variables

Shells support variables. They may be assigned values, manipulated and used. Convention dictates that the variables used in scripts are in UPPERCASE.
The script below shows how to assign and use a variable. In general, shell variables are all treat as *strings* (i.e. bits of text). Shells are extremely fussy about getting the syntax exactly right; in the assignment there must be no space between the variable name and the equals sign, or the equals sign and the value. To use the variable, it is prefixed by a dollar '$' character.

```
#!/bin/ksh
# NAME is a variable
NAME="fred"
echo "The name is $NAME"
```

The special variables $1-$9 correspond to the arguments passed to the script when it is invoked. For example, if we rewrite the script above as shown below, calling the script **name**, and then invoke the command **name Dave Smith**, the message "Your name is Dave Smith" will be printed out:

```
#!/bin/ksh
echo "Your name is $1 $2"
```

Arithmetic in Shell Scripts

Shell scripts can also do arithmetic, although this does not come particularly naturally to them. The script below adds one to the number passed to it as an argument. To do this, it must use the **expr** command, enclosed in back-quote characters. Once again, precise syntax is critical. You must use the correct type of speech marks and the arguments of the **expr** command (**$1**, **+** and **1**) must be separated by spaces:

```
#!/bin/ksh
RESULT=`expr $1 + 1`
echo "Result is $RESULT"
```

There is an alternative to using the expr command and that is to use the basic calculator (bc). This can be invoked from the command line by typing bc, which places you into the interactive basic calculator (don't expect too much!, it is not very friendly!). You then enter the calculation to be performed eg 2*2 and the result is displayed. To exit simply press cntrl D.

The basic calculator may also be accessed within a shell script by piping the calculation to be performed to it:

```
#!/bin/ksh
N1=2
N2=5
echo "$N1 * $N2"|bc
```

This would cause the result 10 to be writted to standard output. Similarly the result can be placed into a variable:

```
#!/bin/ksh
N1=2
N2=5
RESULT=`echo "$N1 * $N2"|bc`
echo "Result is $RESULT"
```

The basic calculator has the following basic commands:

```
+ - * /
% Integer remainder function eg 11%3 would give 3
^ raise to the power of
sqrt square root function
```

Conditional Statements in Scripts

The IF Statement

The most frequently used conditional operator is the **if-statement**. Before considering the **if** statement, it is necessary to explain something of the condition arrangement.

The **test** command will return an **exit status** (the exit status is zero if the condition is true otherwise, it is non-zero). There are many parameters for the **test** command (see below) for example –d, which causes the *test* command to determine if the following argument is a directory.

Thus **test –d $DIRNAME** will check to see if a directory exists (in the current directory with the name contained in the variable DIRNAME) if it does, then it will return a zero exit status. **CAUTION** if you simply type the above at the command line nothing will appear to happen, this is because you need to tell the Korn shell what to do if the test is TRUE or NOT TRUE. This can be achieved by coupling it with the **if** statement.
Syntax:

```
if (condition)
then
        [commands]
else
        [commands]
fi
```

For example, the shell below displays the contents of a file on the screen using **cat**, but lists the contents of a directory using **ls**:

```
#!/bin/ksh
# show script
if (test -d $1)
then
  ls $1
else
```

```
  cat $1
fi
```

As you will recall, there unfortunately is an alternative way of writing a condition in the Korn shell – the test command may be omitted and the round parenthesis () replaced by square ones [ ]. Thus the following if statement:
if (test -d $1)
may be re-written as
if [-d $1]

NOTE the spaces after the first [and before the second.

There are a number of conditions supported by shell scripts; for a complete list, use the on-line manual on the **test** command (**man test**). Some examples are: **-d** (is a directory?), **-f** (is a file?), **=** (are two strings the same?), **-r** (is string set?), **-eq** (are two numbers equal?), **-gt** (is first number greater than second?). You can also test whether a variable is set to anything, simply by enclosing it in quotes in the condition part of the if-statement. The script below gives an example:

More Samples

```
#!/bin/ksh
# Script to check that the user enters one argument, "fred"
if  (test "$1")
then
  echo "Found an argument to this script"
  if [ $1 = "fred" ]
  then
    echo "The argument was fred!"
  else
    echo "The argument was not fred!"
  fi
else
  echo "This script needs one argument"
fi
```

It is possible to *nest* constructs, which means to put them inside one another.

Here, there is an outer if-statement and an inner one. The inner one checks whether *$1* is "fred", and says whether it is or not. The outer one checks whether *$1* has been given at all, and only goes on to check whether it is "fred" if it does exist. Note that each if-statement has its own corresponding condition, *then*, *else* and *fi* part. The inner if-statement is wholly contained between the *then* and *else* parts of the outer one, which means that it happens only when the first condition is passed.

```
#!/bin/ksh
# join command - joins two files together to create a third
# Three parameters must be passed: two to join, the third to create
# If $3 doesn't exist, then the user can't have given all three
if (test "$3")
then
# this cat command will write out $1 and $2; the  operator redirects
# the output into the file $3 (otherwise it would appear on the screen)
  cat $1 $2 > $3
else
  echo "Need three parameters: two input and one output. Sorry."
fi
```

```
#!/bin/ksh
# An alternative version of the join command
# This time we check that $# is exactly three. $# is a special
# variable which indicates how many parameters were given to
# the script by the user.
if [ $# -eq 3 ]
then
  cat $1 $2 > $3
else
  echo "Need exactly three parameters, sorry."
fi
```

```
#!/bin/ksh
# checks whether a named file exists in a special directory (stored in
# the dir variable). If it does, prints out the top of the file using
# the head command.
# N.B. establish your own dir directory if you copy this!
DIR=$HOME/safe
if [ -f $DIR/$1 ]
then
  head $DIR/$1
fi
```

Case Statements

The **if** condition is suitable if a single possibility, or at most a small number of possibilities, are to be tested. However, it is often the case that we need to check the value of a variable against a number of possibilities. The **case** statement is used to handle this situation. The script below reacts differently, depending on which name is given to it as an argument.
Syntax:

Case variablename in
Option1) commands;;
Option2) commands;;
***) commands;;**
esac

Eg:

```
#!/bin/ksh
# Script Written by Phil Irving and Andrew Hunter 18/6/98
case "$1" in
  fred)
    echo "Hi fred. Nice to see you"
    ;;
  joe)
    echo "Oh! Its you, is it, joe?"
    ;;
  harry)
    echo "Clear off!"
    ;;
  *)
    echo "Who are you?"
    ;;
esac
```

The case-statement compares the string given to it (in this case "$1", the first argument passed to the script) with the various strings, each of which is followed by a closing bracket. Once a match is found, the statements up to the double semi-colon (;;) are executed, and the case-statement ends. The asterix * character matches anything, so having this as the last case provides a default case handler (that is, what to do if none of the other cases are matched). The keywords are case, in and esac (end of case).

Further Input – the Read Command

So far, we have only passed parameters to the shell script from the command line however, it is possible for the script to interact with the user. This is achieved using the read command, which places what the user enters into variables. If users are to type several words or values, **read** can be given a number of arguments.
Syntax:

read var1 var2 var3 ….

Egs

```
#!/bin/ksh
echo "Please enter your name /n"
read NAME
echo "Hello $NAME"
```

```
#!/bin/ksh
echo "Please enter your firstname followed by a space and
your lastname /n"
read FIRSTNAME LASTNAME
echo "Hello $FIRSTNAME $LASTNAME"
```

Quoting in Scripts

You will recall that shells (and also scripts) no less than three different types of quotes are used, all of which have special meanings. We have already met two of these, and will now consider all three in detail.

Two types of quotes are basically designed to allow you to construct messages and strings. The simplest type of quotes are single quotes; anything between the two quote marks is treated as a simple string. The shell will not attempt to execute or otherwise interpret any words within the string.

The script below simply prints out the message: "your name is fred."

```
#!/bin/ksh
echo 'Your name is fred'
```

What happens if, rather than always using the name "fred," we want to make the name controlled by a variable? We might then try writing a script like this:

```
#!/bin/ksh
NAME=fred
echo 'Your name is $NAME'
```

However, this will **not** do what we want! It will actually output the message "Your name is $name", because anything between the quote marks is treated as literal text - and that includes $name.

For this reason, shells also understand double quotes. The text between double quotes marks is also interpreted as literal text, except that any variables in it are interpreted. If we change the above script to use double quotes, then it will do what we want:

```
#!/bin/ksh
NAME=fred
echo "Your name is $NAME"
```

The above script writes out the message: "Your name is fred." Double quotes are so useful that we normally use them rather than single quotes, which are only really needed on the rare occasions when you actually want to print out a message with variable names in it.

The third type of quotes are called back-quotes, and we have already seen them in action with the **expr** command. Back-quotes cause the Shell to treat whatever is between the quotes as a command, which is executed, then to substitute the output of the command in its place. This is the main way to get the results of commands into your script for further manipulation. Use of back-quotes is best described by an example:

```
#!/bin/ksh
TODAY=date
echo "Today is $TODAY"
```

The **date** command prints out today's date. The above script attempts to use it to print out today's date. However, it does not work! The message printed out is "Today is date". The reason for this is that the assignment **today=date** simply puts the string "date" into the

variable today. What we actually want to do is to execute the **date** command, and place the *output* of that command into the **today** variable. We do this using back-quotes:

```
#/bin/ksh
TODAY=`date`
echo "Today is $TODAY"
```

Back-quotes have innumerable uses. Here is another example. This uses the grep command to check whether a file includes the word "and."

```
#!/bin/ksh
# Check for the word "and" in a file
RESULT=`grep and $1`
if [ "$RESULT" ]
then
  echo "The file $1 includes the word and"
fi
```

The *grep* command will output any lines in the file which do include the word "and." We assign the results of the *grep* command to the variable *result*, by using the back-quotes; so if the file does include any lines with the word "and" in them, *result* will end up with some text in it, but if the file doesn't include any lines with the word "and" in them, *result* will end up empty. The if-statement then checks whether result has actually got any text in it.