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 Bash-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/bash
echo "hello world"
```

Execute the script (./filename) 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 (bash 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/bash
# 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/bash
# 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/bash
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/bash
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/bash
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/bash
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/bash
# 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/bash
# 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/bash
# 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/bash
# 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/bash
# 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/bash
# 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/bash
echo "Please enter your name /n"
read NAME
echo "Hello $NAME"
```

```
#!/bin/bash
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/bash
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/bash
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/bash
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/bash
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/bash
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/bash
# 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.

Conditional Statements - Exercise 1

Write a shell script to accept a name passed to it as a parameter. If the name exists, then output hello followed by the name.
Modify the above script to give a suitable error message if the name is not supplied.
Modify the above script to determine that if the name exists that it is John.
Write a shell script to accept two numbers from the command line, compare them and determine which is the larger. It should output a suitable message (e.g. The first number, X, is greater than the 2nd number Y). X and Y represent the parameters passed in.
Modify the above script to check whether the numbers exist before commencing the test.
Try entering the following shell scripts and note their output. Write a short user guide for these utitlities and provide sample input and expected output.

Conditional Statements Continued

Case Statement - Exercises

Write a shell script to accept the name of your friend passed at the command line, and issue an appropriate greeting to that friend. Save this as casefriend
Modify the above script to determine whether the parameter exists before determining your friends name and giving the greeting. Save this as casefriend2
Write a menu shell script which will take a parameter from the command line and tell the user the option they have chosen. We will modify this later to become more interactive.

Conditionals - Review Exercises

1) Write a script called **fileexists**, which indicates whether a named file exists or not.
2) Write a modified version of the **show** example script (the first example of the if statement given above), which prints out the message "File does not exist" if the user gives a name which isn't a file or directory, and the message "You must give an argument" if the user doesn't give an argument to the program.
3) Write a script called **save** which copies a file into a special directory, and another called **recover** which copies a file back out of the special directory. The user of the script should not be aware of the location of the special directory (obviously the script will be).
4) Alter your scripts so that, if you try to save a file which already exists in the special directory, the script refuses to save the file and prints out a message to that effect.

Further Input - the read command

Further Input - the read command exercises

Write a shell script to input two numbers from the user (using the read command) and multiply them together.
Modify the above script to divide (/) the two numbers
Modify the above script to add the two numbers.
Modify the above script to subtract the two numbers.
Modify your menu program created earlier to accept input during execution (using read) rather than from the command line.

Further Conditional Commands: the select statement

The select statement may be used to quickly generate a menu - it simpl;y produces a numbered list on the standard output from which the user can select.

Syntax:
Eg:

```
select REPLY in "casefriend1" "casefriend2"
do
case $REPLY in
"casefriend1")
```

```
casefriend1;;
"casefriend2")
casefriend2;;
*)
echo "Wrong choice";;
esac
done
```
Although this seems a quick way to generate menus, it is not as elegant as an "echo .. case" solution.

Further Conditional Commands: the select statement Exercise

Implement the above example

Quoting in Scripts

Quoting in Scripts - Exercises.

Write shell scripts for the above examples and try them out.
Write a script which checks whether a given file contains a given word. If it does, the script should output the message "The file contains the word"; if not, it should output the message "The file doesn't contain the word.". Save this as wordsearch.
Modify the above script to determine whether a parameter was passed before commencing the grepping process.

Modular Programming in Shell Scripts

Like any good programming language, shells allow modular programming. There are two basic forms of modular programming in shell scripts:
By calling sub shells
Functions

Calling Sub shells

Possibly unbeknown to you, you actually undertook this in the variable tutorial to experiment with variable inheritence. To call a sub shell, you simply enter its name on a line of the shell script (as you would any other UNIX command). For example, let us suppose we wish to call the today script (created in the above examples) after we have given the present working directory, we would use:

```
pwd
today
```

Should parameters require to be passed (eg in wordsearch), then these can be passed by placing them on the command line:

```
pwd
today
wordsearch and andfile
```

The above script will call the wordsearch script and pass to it the values and (as the search word) and andfile (as the file to be searched). Alternatively, these parameters could themselves, be replaced by variables eg:

```
pwd
today
wordsearch $SEARCHWORD $SEARCHFILE
```

Functions

Will be dealt with later in the tutorial.

Modular Programming in Shell Scripts - Exercises

Write a shell script which calls the today shell script.
Write a scripts which calls the name script.

Looping Commands - while

Whereas conditional statements allow programs to make choices about what to do, looping commands support repetition. Many scripts are written precisely because some repetitious processing of many files is required, so looping commands are extremely important.

The simplest looping command is the **while** command. The syntax of which Iis given below:

while (condition)
do
[commands]
done

Eg:

```
#!/bin/bash
# Script Written by Phil Irving and Andrew Hunter 18/6/98
#Output my intention followed by a new line
echo "I am going to count up to 10\n"
#start a 1
NUMBER=1
#while number is less than 12
while (test $NUMBER -le 10)
do
#print out the number
      echo $NUMBER
#add 1 to the number
      NUMBER=`expr $NUMBER + 1`
done
```

The above script repeats the while-loop 10 times; with the number stepping through from 1 to 10. The body of the loop is enclosed between the **do** and **done** commands. Every time the **while** command is executed, it checks whether the condition in the square brackets is true. If it is, then the body of the while-loop is executed, and the computer "loops back" to the **while** statement again. If it isn't, then the body of the loop is skipped.

If a while-loop is ever to end, something must occur to make the condition become untrue. The above example is a typical example of how a loop can end. Here, the number variable is initially set to one. Each time through the loop it is incremented (i.e. has one added to it); once it reaches 10, the condition fails and the loop ends. This is the standard technique for repeating something a set number of times. If you get inadvertently stuck in such a loop, you can always press Ctrl-C to break out.

Just as if statements can be nested, so can while statements. In the example below the script causes all the "times-tables" to be printed up to 12:

```
#!/bin/bash
# Script Written by Phil Irving and Andrew Hunter 18/6/98
#Output my intention followed by a new line
echo "I am going to print out the times tables - from 1 to 12\n"
#set the outer loop at 1
OUTER=1
#The OUTER loop
while (test $OUTER -le 12)
do
#Set the inner loop at 1 - dont forget the inner loop must be set to one each
#   time we pass through the outer loop
INNER=1
echo " "
echo "$OUTER times table"

#The INNER loop
      while (test $INNER -le 12)
            do
#Working out the value of inner times outer
#The \* is the syntax for multiplication with the expr command
                  RESULT=`expr $INNER \* $OUTER`
                  echo "$INNER x $OUTER = $RESULT"
```

```
#Incrementing the inner count by 1
                INNER=`expr $INNER + 1`

            done
#Incrementing the outer loop by 1
OUTER=`expr $OUTER + 1`
done
```

Looping Commands - while exercises

Write a shell script which will count from 1 to 100.
Modify the above script to count backwards from 100 to 1
Write a shell script to produce "times tables" up to 60.
Modify the above to produce the times table in reverse (60 to 1).
Modify your menu program to repeat until the user enters 9 (for exit). Hint you may need to include an option 9 in your case statement.

Looping Commands - for

Another form of looping command, which is useful in other circumstances, is the **for** command. The **for** command sets a variable to each of the values in a list, and executes the body of the command once for each value.The syntax is given below:

for var in [list]
do
commands
done

Eg (simple):

```
#!/bin/bash
# Script Written by Phil Irving and Andrew Hunter 18/6/98
for NAME in fred joe harry
do
  echo "Hello $NAME"
done
```

The script above prints out the messages "Hello fred," "Hello joe," and "Hello harry." The command consists of the keyword for, followed by the name of a variable (in this case, $name, but you don't use the dollar in the for-statement itself), followed by the keyword in, followed by a list of values. The variable is set to each value in turn, and the code between the do and done keywords is executed once for each value.

The for-loop is most successful when combined with the ability to use wildcards to match file names in the current directory. The for-loop below uses the * wildcard to match all files and sub-directories in the current directory. Thus, the loop below is executed once for each file or directory, with $file set to each one's name. This script checks whether each one is a directory, using the -d option, and only writes out the name if it is. The effect is to list all the sub-directories, but not the files, in the current directory.

```
#!/bin/bash
# Script Written by Phil Irving and Andrew Hunter 18/6/98
for FILE in *
do
  if [ -d "$FILE" ]
  then
    echo "$FILE"
```

```
  fi
done
```

Looping Commands - for Exercises

Write a script to output (to standard output) 4 files named in the script (ensure the four files all begin with the prefix bu1).
Modify the above script to automatically output any files in the current directory which begin with the prefix bu1.
Create a directory called bupdir and another called copydir. Write a script which will copy files from the copydir into the bupdir and save it as backup. Hint your script must work irrespective of its own location (and therefore paths inside the script will need to be spcified using environmental variables).

Looping Commands - until

The third looping command available to us is until loops. These are very similar to the while loop except that it executes until the condition is true.

Syntax:
Eg:

```
#!/bin/bash
# Script Written by Phil Irving and Andrew Hunter 18/6/98
#Output my intention followed by a new line
echo "I am going to count up to 10\n"
#start a 1
NUMBER=1
#until number is greater than 12
until (test $NUMBER -gt 12)
do
#print out the number
      echo $NUMBER
#add 1 to the number
      NUMBER=`expr $NUMBER + 1`
done
```

Looping Commands - until Exercises

Write a script to count backwards from 100 to 1.
Modify the above to count forwards.
Rewrite the times tables exercises (from the while command) to work with the until command.
Modify the above to produce the times table in reverse (60 to 1).
Modify your menu program (previously modified in the while exercises).

Looping Commands - Review Exercises

Write a menu program which allows the user to select form the following:
Times table from 1 to 12
Times table from 1 to 60
Times table form 12 to 1
Times table from 60 to 1
Add two numbers together you specify
Add together X numbers you specify (you also tell me X)
Ascending times table from X to Y (you specify X and Y)
Descending times table from X to Y (you specify X and Y)
Ascending times table from X to Y (you specify X, Y and I (increment))

Descending times table from X to Y (you specify X, Y and I (increment))
You must write all the associated shell scripts
Modify the above script so that the increment is specified in the calling shell and inherited by the sub shell.

Consolidation Exercises

Alter your **save** script so that, if a file has previously been saved, the user is asked whether it should be overwritten (Hint: use the **read** command to get the user's decision).
Write a script which lists all files in the current directory which have also been stored in your special directory by the **save** script.
Adapt your answer to exercise five so that you list all files in your current directory which include a given word.
Write an interactive script which allows the user to repeatedly apply the **save** and **restore** scripts. It should continuously prompt the user for commands, which can be either of the form **save <file**, **restore <file**, or **quit**. Hint: use **while**, **read** and **case**.
Right a script which searches all the sub-directories of your current directory for files with a given name. Hint: use for, if, cd.

More Sample Scripts

Below is a more complex script, which acts as a DOS command interpreter. DOS uses the commands **cd, dir, type, del, ren** and **copy** to do the same functions as the UNIX commands **cd, ls, cat, rm, mv** and **cp**. This script loops continuously, allowing the user to type in DOS commands, which are stored in the variables **$command**, **$arg1** and **$arg2**. The command is considered by the case statement, which executes an appropriate UNIX command, depending on which DOS command has been given:

```
#!/bin/bash
# Script Written by Phil Irving and Andrew Hunter 18/6/98
# DOS interpreter. Impersonates DOS as follows:
# DOS command   UNIX equivalent   Action
# dir           ls                List directory contents
# type          cat               List file contents

echo "Welcome to the DOS interpreter"
echo "Type exit to leave"
#Setting the variable command to ensure the loop operates
COMMAND=Notexit
# loop
while (test $COMMAND != "exit")
do
# Show DOS prompt; \c stops a new line from being issued
  echo "DOS \c"
# Read in user's command
  read COMMAND ARG1 ARG2
# Do a UNIX command corresponding to the DOS command
  case $COMMAND in
    type)
      cat $ARG1
```

```
    ;;
  del)
    rm $ARG1
    ;;
  exit)
    # The leave option therefore do nothing
    ;;
  *)
    echo "DOS does not recognise the command $COMMAND"
    ;;
 esac
done
```

More Sample Scripts - Exercises

Implement the DOS command interpreter given above.
Once you have it operational, write a user guide to explain to a novice user how the system works.

Aliasing

In the above exercise, we created a command interpreter for DOS which made UNIX emulate DOS. This works fine whilst running the script but doesn't allow users to use any UNIX commands or call shell scripts. There is an alternative however, we can alias commands to be something else. For instance we can tell the Korn shell that whenever someone types dir to execute the ls -l command. We do this by creating an alias for dir:
alias dir='ls -l'
From now on, whenever we type dir the ls -l command will run (until we log out). Commands themselves can also be aliased. For instance if we don't want anyone to have access to the talk command we can simply alias it to null:
alias talk='null'
If this is included in a startup script, then it simply stops the user being able to access the talk command (by effectively hiding it). Of course the systems administrator could also remove user privileges to that command.
To remove an alias, we use the unalias command thus to remove the talk alias, we would use:
unalias talk

Aliasing Exercises

Alais the dir command
Unalias the dir command
Alias the talk command
Unalias the talk command
Try implementing the DOS interpreter as alias commands.
Unalias these commands

Functions

The Korn shell supports functions in a similar way to C. Within a script, a function may be declared and sunsequently used. It may also have parameters passed to it (in a similar fashion to C).
Syntax:
[function] name
{
list of commands
}
Eg:

```
#!/bin/bash
# Script Written by Phil Irving and Andrew Hunter 18/6/98
#Function Declaration without parameters
greeting ()
{
echo hello
echo world
}
#Set count variable
ITCOUNT=1
#Iteration to run funciton 4 times
while [ ITCOUNT -le 4 ]
do
#Call greeting function
greeting
ITCOUNT=`expr $ITCOUNT + 1`
done
```

Function may also use parameters in a similar fashion to calling a script, ie by the positional method

Eg:

```
#!/bin/bash
# Script Written by Phil Irving and Andrew Hunter 18/6/98
# Script to demonstrate use of functions in shell scripts
mynamefunc1 ()
{
echo "My name is $NAME" # display the contents of variable
NAME
echo "All parameters are $* # display all parameters in the
list
}
# Main part of the script
# Calling script
# with one parameter
mynamefunc1 Phil
# with many
mynamefunc1 Phil Bill Geoff
```

Returning from a Function

To return from a function, you may either leave as above or you can use the **return** command. By using the return command, it is possible to control what is returned from the function. If for instance we want the function to add two numbers together and return the

value back to the main part of the script, we can do so by setting the exit code of the function to a value (be careful if you are testing the variable for true - you could get some interesting results!). The exit code is accessible in the main part of the script by using the #? Variable.
Things to note here: if no argument is used, the function exits with the exit code set to the exit code of the last command executed
Eg:
#!/bin/bash

```
# Script Written by Phil Irving and Andrew Hunter 18/6/98
# This script adds two values in a function and returns the
result
# using the exit value
addnumfunc ()
{
FUNCRESULT=`expr $1 + $2`
return FUNCRESULT
}
# Main part of the script
addnumfunc 3 7
SCRIPTRES=$?
echo "The result is $SCRIPTRES"
```

It is perfectly possible to write recursive functions, it just needs some careful thought. Below is an example to implement a recursive factorial script:
#!/bin/bash

```
# Script Written by Phil Irving and Andrew Hunter 18/6/98
# This script calls a function recursively to solve factorial
for a number
# input
facfunc () # factorial function
{
if [ $1 -le 1] # if a value of 1 or less is entered we dont
need to solve
then
return 1 # return a value of 1
else
FUNCTEMPVAR=0 # create two temporary variables
FUNCRESULT =0
FUNCTEMPVAR=`expr $1 - 1` # subtract one from the number sent
facfunc $FUNCTEMPVAR # call the function recursively
FUNCRESULT=`expr $? * $1` # multiply the result returned from
the
# fuction by the value sent
return $FUNCRESULT # return the result
fi
}
# Main script
echo "Please enter a positive value for factorial \n"
read INPUTFAC
funcfac $INPUTFAC
echo "The factorial of $INPUTFAC is $?"
```

Functions - Exercises

Try out the examples given above, save first as funcfour, the second as funcparams, the third as funcadd, and the final one as recfunc.
Modify the funcfour script to run 10 times
Modify the funcadd script to multiply the numbers and save as funcmul.
Modify the funcadd script to request from the user the number of parameters to be added, receive them and then add them together.
Write a script which implements a single times table by requesting from the user a value for the times table, an incremental value, and a stop value. Hint you should inplement a loop of your choice, your script should then call a function (from inside the loop) over again (sending it both the cumulative total and the value to add) until the loop terminates.

Korn Shell Summary

Variables

Variables are assigned using the equals sign, no spaces. They are used by preceding them with a dollar character.

Arguments

Arguments are labelled $1, $2, ..., $9. $# indicates how many arguments there are. **shift** moves all the arguments down, so that $2 becomes $1, $3 becomes $2, etc. This allows more than nine arguments to be accessed, if necessary.

Quotes

Single quotes ('string') form string literals. No interpretation is performed on the string.

Double quotes ("string") form string literals with limited substitution: variables are replaced with their value, and back-quoted commands are replaced with the results of their execution. A backslash '\' at the end of a line allows strings to be stretched over a number of lines.

Back quotes (`string`) execute the string in a sub-shell, and substitute in the results of the execution.

Conditionals

if [<condition] then ... elif ... else ... fi. **elif** and **else** are optional.

case <string in <case1) ... ;; <case2) ... ;;; esac. The case ***)** acts as a default for any value not matched by one of the earlier cases.

Looping

for <variable in <list do ... done

while [<condition] do ... done

until [<condition] do ... done

Expressions

The **expr** command will do calculations. It usually needs to be enclosed in back-quotes, as the result of the calculation will be assigned to some variable. The arguments to the **expr** command *must* be separated by spaces. The value of **`expr 3 + 1`** is "**4**", whereas the value of **`expr 3+1`** (no spaces) is the string "**3+1**".

Input

User-input can be solicited using the **read** command, which places what the user types into variables. If users are to type several words, **read** can be given a number of arguments.

Functions

Can be called from within a shell script. By using the return command, it is possible to return a value to the main script (such as a result).

Review Exercises

Modify the DOS command interpreter above to work with the following DOS commands: cd, dir, copy and rename
Write a menu program which runs iteratively offering the user the following choices:
Times table menu
DOS command emulator
Exit to UNIX
The user should be able to enter their choice and your system should call the appropriate subshell. The timestable menu should be the system you created in the looping exercises.
You have written a number of scripts in this tutorial. Review each of the scripts that you have written and ensure that they contain enough comments to: allow you to be able to understand them after a period of 6 months; or for a novice to be able to pick up the scripts and be able to appreciate what each line does.
For each of the conditional commands we have discussed in this tutorial, append to your user manual a description of the command together with appropriate examples. You description should also justify when you would use the different commands.
For each of the looping commands we have discussed in this tutorial, append to your user manual a description of the command together with appropriate examples. You description should also justify when you would use the different commands.
For each of the methods of modular programming we have discussed in this tutorial, append to your user manual a description of the method together with appropriate examples. You description should also justify when you would use the different commands. Modify the funcparams to pass the parameters one at a time (by using a loop of your choice). You must include a detailed description of the workings of the program and justify why you have undertaken the task in this manner.
Write a script containing a function to request from the user a temperature in either Fahrenheit or centigrade and then perform the calculation to the other (hint you will need to make a decision based upon whether the temp is F or C). You must include a detailed description of the workings of the program and justify why you have undertaken the task in this manner.
Write a script which will list all of the files in a users directory and add all the filesizes together - it should display all of the filenames and size followed by a summary line stating

"total size is". Hint you will need to pipe a directory listing to a file and then cut out the relevant field, and call the function with the latest value of the cumulative total. You must include a detailed description of the workings of the program and justify why you have undertaken the task in this manner.