Writing UNIX Scripts

Introduction

In UNIX, commands are submitted to the Operating System via a *shell*. A shell is an environment which allows commands to be issued, and also includes facilities to control input and output, and programming facilities to allow complex sets of actions to be performed. Whenever you type commands at the prompt in Unix, you are actually communicating interactively with a shell.

In addition to typing commands directly to the shell, you can place them in a file (which must be given execute permission), and then run the file. A file containing UNIX shell commands is known as a *script* (or shell script). Executing the script is equivalent to typing the commands within it.

The Unix shells are actually quite sophisticated programming languages in their own right: there are entire books devoted to programming in them. Together with the large number of special utility programs which are provided as standard, scripts make Unix an extremely powerful operating system.

Scripts are *interpreted* rather than *compiled*, which means that the computer must translate them each time it runs them: they are thus less efficient than programs written in (for example) C. However, they are extremely good where you want to use Operating System facilities; for example, when processing files in some fashion.

There are actually no less than three different types of scripts supported in Unix: Bourne shell, C shell, and Korn shell. Bourne is the most common, Korn the most powerful, and C the most C-like (handy for C programmers). This tutorial will concentrate on the simplest of the three: the Bourne shell.

A simple Bourne-shell script

If you simply type Unix commands into a file, and make it executable, then run it, Unix will assume that the commands in it are written in whatever shell language you happen to be using at the time (in your case, this is probably the C shell). 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 Bourne-shell script: it just outputs the message "hello world":

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

Use the text editor to create a file with the lines above in it. Save the file, calling it *hello*. Then make the *hello* file executable by typing:

```
chmod 755 hello
```

You can then run the script by simply typing *hello*. When you type hello, UNIX realises that this is a script, and runs the UNIX commands inside it (in this case, just the *echo* command).

Naming Shell Script Files

You can give your shell scripts almost any name you like. Be careful, however! If the name you use

happens to already be an existing UNIX command, when you try to run your script you will end up running the system-defined command instead. Some fairly obvious names can cause you this problem; for example, *test*. To make sure that your intended name is O.K., you should check whether it exists before you start editing the new command. You can do this by using the *which* command, which will locate a command, if it exists, and tell you that it can't locate it if it doesn't. For example, to find out whether there are commands called *dc* and *dac*, type:

```
which dc
which dac
```

Most UNIX commands are stored in the */bin* directory, so you can get a list of most of them by typing:

```
ls /bin
```

Comments and Commands

In the Bourne shell, any line beginning with a hash '#' character in the first column is taken to be a comment and is ignored. The only exception is the first line in the file, where the comment is used to indicate which shell should be used (**/bin/sh** is the Bourne shell). It is always good practice to use comments to indicate what a script does; both in a header section at the top, and line by line if the code is at all complicated. There aren't many comments in the examples in this document, to reduce the amount of paper used in printing it. Your scripts should be far more liberally commented.

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/sh
# purpose: print out current directory name and contents
pwd
ls
```

Shell Variables

Like every programming language, shells support variables. Shell variables may be assigned values, manipulated, and used. Some variables are automatically assigned for use by the shell.

The script below shows how to assign and use a variable. In general, shell variables are all treated 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/sh
# 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/sh
echo "Your name is $1 $2"
```

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, as they have a special meaning (this will be explained later), and the arguments of the **expr** command (**$1**, **+** and **1**) must be separated by spaces.

```
#!/bin/sh
result=`expr $1 + 1`
echo "Result is $result"
```

Conditionals in Shell Scripts

All third-generation programming languages share a number of critical features: sequential execution, use of variables, logical and arithmetic operators, conditional branching and looping. This section is concerned with conditional execution.

Conditionals are used where an action is appropriate only under certain circumstances. The most frequently used conditional operator is the *if-statement*. 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/sh
# show script
if [ -d $1 ]
then
  ls $1
else
  cat $1
fi
```

Here, we notice a number of points:

- The if-statement begins with the keyword **if**, and ends with the keyword **fi** (**if**, reversed).
- The **if** keyword is followed by a *condition*, which is enclosed in square brackets. In this case, the condition **-d $1** may be read as: *if $1 is a directory*.
- The line after the **if** keyword contains the keyword **then**.
- Optionally, you may include an **else** keyword.

If the condition is satisfied (in this case, if **$1** is a directory) then the commands between the **then** and **else** keywords are executed; if the condition isn't satisfied then the commands between the **else** and **fi** keywords are executed. If an **else** keyword isn't included, then the commands between the **then** and **fi** keywords are executed if the condition is true; otherwise the whole section is skipped.

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:

```
#!/bin/sh
# Script to check that the user enters one argument, "fred"
```

```
if [ "$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
```

The above script illustrates another important feature of if-statements, and this is true of all the other constructs covered in this Guide. 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.

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.

```
#!/bin/sh
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).

Examples

```
#!/bin/sh
# 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 [ "$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/sh
# 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/sh
# 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/cs0ahu/safe
if [ -f $dir/$1 ]
then
  head $dir/$1
fi
```

Exercises

1) Write a script called **exists**, 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.

Quoting in Scripts

Confusingly, in Shell 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/sh
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/sh
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/sh
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 rate 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/sh
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/sh
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/sh
# 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.

Exercises.

5) 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."

6) **[Optional - tricky!]** Write a script which checks whether a given file was created today (Hint: the **ls -l** command includes a listing of the date when a file was created).

```
#!/bin/sh
today=`date`
result=`grep $today $1`
if [ "result" ]
then
  echo "The file $1 was created today"
else
  echo "The file $1 was not created today"
fi
```

Looping Commands

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. An example is given below:

```
#!/bin/sh
# Start at month 1
month=1
while [ $month -le 12 ]
do
# Print out the month number
  echo "Month no. $month"
# Add one to the month number
  month=`expr $month + 1`
done
```

echo "Finished"

The above script repeats the while-loop twelve times; with the month number stepping through from 1 to 12. 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 month variable is initially set to one. Each time through the loop it is incremented (i.e. has one added to it); once it reaches 12, the condition fails and

the loop ends. This is the standard technique for repeating something a set number of times.

Occasionally, it can actually be useful to loop unconditionally, but to break out of the loop when something happens. You can do this using a **while** command with a piece of text as the condition (since the piece of text is always there), and a **break** command to break out of the loop. The computer will go round and round the loop continuously, until such time as it gets to the **break** statement; it will then go to the end of the loop. The break statement is issued from within an if-statement, so that it only happens when you want to loop to end. The example below loops continuously until the user guesses the right word. If you get inadvertently stuck in such a loop, you can always press Ctrl-C to break out.

This example also demonstrates how a shell script can get input from the user using the **read** command. The script loops continuously around the while-loop, asking the user for the password and placing their answer in the **answer** variable. If the **answer** variable is the same as the **password** variable, then the **break** command breaks out of the loop.

```
#!/bin/sh
password="open"
answer=""
# Loop around forever (until the break statement is used)
while [ "forever" ]
do
# Ask the user for the password
  echo "Guess the password to quit the program> \c"
# Read in what they type, and put in it $answer
  read answer

# If the answer is the password, break out of the while loop
  if [ "$answer" = "$password" ]
  then
    break
  fi
done
# If they get to here, they must've guessed the password,
# because otherwise it would just keep looping
echo "Good guess!"
```

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. A simple example is given below:

```
#!/bin/sh
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/sh
for file in *
```

```
do
  if [ -d "$file" ]
  then
    echo "$file"
  fi
done
```

Exercises

6) 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).

7) Write a script which lists all files in the current directory which have also been stored in your special directory by the **save** script.

8) Adapt your answer to exercise five so that you list all files in your current directory which include a given word.

9) 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**.

10) 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 give.

```
#!/bin/sh
# DOS interpreter. Impersonates DOS as follows:
# DOS command   UNIX equivalent   Action
# cd            cd                Change directory
# dir           ls                List directory contents
# type          cat               List file contents
# del           rm                Delete a file
# ren           mv                Rename a file
# copy          cp                Copy a file

echo "Welcome to the DOS interpreter"
echo "Type Ctrl-C to exit"

# Infinite loop
while [ "forever" ]
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
    cd)
      cd $arg1
      ;;
    dir)
      ls
```

```
      ;;
    type)
      cat $arg1
      ;;
    del)
      rm $arg1
      ;;
    ren)
      mv $arg1 $arg2
      ;;
    copy)
      cp $arg1 $arg2
      ;;
    *)
      echo "DOS does not recognise the command $command"
      ;;
  esac
done
```

Bourne 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

You can break out of a loop using the **break** command; you can jump to the beginning of the loop and begin its next execution using the **continue** command.

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.