SHELL PROGRAMMING

- Shell Scripts

- When a group of commands have to be executed regularly,they should be stored in a file and the file itself is executed as a shell script.

- Shell scripts are executed in a separate child shell process,and this sub-shell need not be same type as your login shell.

- We mentioned that shell scripts are executed by a child shell.You can also explicitly spawn a child of your choice with the script name as argument.

$sh script.sh

Read statement

- The read statement is the shell's Interpretive tool for taking input from the user i.e making scripts interactive.It is used with one or more variables.
- Input supplied through the std input is read into these variables when you use a statement like
- $read name
- Since this is the form of assignment no $ is used before name.
- Eg echo "Enter the pattern to be searched:\c"
- read pname
- echo "Enter the file to be used
- read flname

Read Statement

- A single read statement can be used with one or more variables to let you enter multiple arguments
- Eg read pname flname
- If the number of arguments supplied is less than the number of variables accepting them, any leftover variables will simply remain unassigned.

However when the number of arguments exceeds the number of variables, the remaining words are assigned to the last variable.

SHELL PROGRAMMING

- We can run the shell scripts run non-Interactively and be used with redirection and pipelines.
- When arguments are specified with a shell script ,they are assigned to certain special variables rather positional parameters.

. The positional parameter starts by $1,the second argument is read is read into

$2 and so on. In addition to these positional parameters, there are few other special parameters used by the shell.Their significance is noted below.

Positional Parameters

- The Positional parameters starts by $1,the second argument is read into $2 and so on.
- In addition to these positional parameters there are few other special parameters used by the shell.Their significance is noted below.

- $*- It stores the complete set of positional parameters as a Single string.
- $# - It is set to the number of arguments specified.This lets you design scripts that check whether the right number of arguments are specified.

- $0 Holds the command name Itself.You can link a shell script to be invoked by more than one name.The script logic can check $0 to behave differently Depending on the name by which it is invoked.

Positional Parameters

- $1,$2 Positional parameters representing command line arguments.
- “$@”: Each quoted String treated as a Separate argument.
- $? Exit status of last command.
- $$ PID of the current shell.
- $! PID of the last background job.

Positional Parameters

- $? Exit status of Last Command
- $* It stores the complete set of positional parameters as a single string.
- $$ PID of the current shell.
- $# It is the set to the number of arguments specified .This let's you design scripts that check whether the right number of arguments are specified.

- Exit and Exit status of a Command
- The command is generally run with a numeric argument
- $exit 0 Used when everything went fine.
- $exit 1 Used when something went wrong.

Eg Shell Program

- echo “Enter the pattern to be searched :\c”
- read pname
- echo “Enter the File to be used: \c”
- read flname
- echo “Searching for $pname from file $flname
- grep “$pname” $flname
- echo “Selected Records selected in filename”

Positional Parameters

- These are two very common exit status or values .It's quite often used with a command when it fails.
- Once grep could'nt locate a pattern we said that the command failed.What we meant was that the exit Function in the grep code was invoked with a non-zero argument[exit(1)]
- This value is communicated to the calling program usually the shell.
- It's through the exit command or Function that every command return's an exit status to the caller.
- Further, a command is said to return a true exit status if it executes successfully and false if it fails.The Shell offers a variable ($?) and a command that evaluates a command exit status.
-

The Parameter $?

- The parameter $? Stores the exit status of the last command.It has the value 0,if the command succeeds and a non-zero value if it fails.
- This parameter is set by the exit's argument If no exit status is specified ,

then $? Is set to zero(true).

 Try using grep in these ways and you'll see it returning three different exit values.

$grep director emp.lst > /dev/null or ; echo $?

$echo $?

0 ------ . Success

$grep director emp.lst > /dev/null or ; echo $?

$echo $? 1---- Failure

The Parameter $?

- The exit status is extremely important for programmers. They use it to devise program logic that branches into different paths depending on the success or failure of a command.

Tip: To find out whether a command executed successfully or not,Simply use the echo $? after the command.

0 indicates success; Other values point to failure.

Eg $cat foo cat can't open foo

This command returns a nonzero exit status because it could'nt open the file.

The shell offers a variable ($?) and a command that evaluates a command exit status.

Logical Operator && and ||

- cmd1 && cmd2
- Cmd1 || cmd2
- The shell provides two operators that allow conditional execution –the

&& and || which typically have the syntax.

- The && delimits two commands .The cmd2 is executed only when cmd1 is executed .The || operator plays an inverse role ;the second command is executed only when the first fails.
- Eg grep ‘director’ emp.lst && echo “pattern found in file”.
- grep ‘manager’ emp.lst || echo “Pattern not found”.

The If Conditional Construct in Shell Program

- The if Statement uses the following forms,much like the one used in other languages.

- if command is successful
- then
- execute commands
- Else
- execute commands
- fi

if command is successfull
then
execute commands
fi

If Conditional Construct in Shell Program

- If command is successful
- then
- execute commands
- else if command is successful
- then ……
- else ……
- fi

Using test and [] to Evaluate Expressions

- When u use if to evaluate expressions you need the test statement because the true or false values returned by expressions can't be directly handled by if test use certain operators to evaluate the condition on its right and returns either a true or false exit status,which is then used by if for making decisions.

- test works in 3 ways
- Compute two numbers
- Compares two strings or single one for a null value.
- Checks a File's Attributes.

If Conditional Construct in Shell Program

- #!/bin/sh
- echo “Enter Source and Target Filenames”
- read source target
- if cp $source $target
- then
- echo “File copied Successfully”
- else
- echo “Copying of File Failed”
- fi

Using test and [] to Evaluate Expressions

- test uses certain operators to evaluate the condition on its right and returns either a true or false exit status,which is used by if for making decisions.
- test works in three ways
- compares two numbers
- compares two strings or a single one for a Null value.
- check a file's attributes.
- These tests can be made by test in association with the shell's other statement also,but for the present we'll stick with if.
- test does'nt display any output but simply sets the positional parameter $?.

Using test and [] to Evaluate Expressions

- test’s does’nt display any output,but simply sets the parameter $?.
- Numeric Comparision

Operator	Meaning
-eq	equal to
-ne	Not equal to
-ge	greater than equal to
-gt	greater than
-lt	lesser than
-le	Lesser than equal to

Using test and [] to Evaluate Expressions

- #!/bin/sh
- echo “Enter a Number from 1 to 10”
- read num
- if test $num –lt 6
- then
- 	echo “I used to think I was indecisive”
- 	 echo “Anonymous”
- fi
- When you use if to evaluate expressions ,you need the test statement because the true or false values returned by expressions can’t be directly handled by if.

Numeric Comparision’s

- The Numeric comparison operators used by test have a form different from what you would have seen anywhere.
- This always begin with a –(hyphen) followed by two letter string,and be enclosed on either side by whitespace -ne Not equal.
- The operator’s are quite mnemonic ; -eq implies equal to –gt implies greater than and so on.Numeric comparision in the shell is confined to integer values only;decimal values are simply truncated.
- $ x=5;y=7;z=7.2
- $ test $x –eq $y ;
- $echo $?
- test $x –lt $y
- echo $?

$ test $z –eq $y ;

$ echo $?

Shorthand For Test Operator

- test is widely used that fortunately there exists a shorthand method of executing it.
- A pair of rectangular brackets enclosing the expression can replace it.Thus the following two forms are equivalent.
- test $x -eq $y
- [$x -eq $y]
- Note that you must provide whitespace around the operators (like –eq)

their operands (like $x) and inside the [and].

.Note: It is a feature of most programming languages that you can use a condition like if(x), where x is a variable or expression.If x is greater than 0,the statement is said to be true,we can also apply the same logic here and use

If [$x] as a shorthand form of if [$x –gt 0]

Test Operator

- >Having used tests as standalone feature You can use it as if's control command .The next script emp3a .sh uses test in an if-elif-else-fi construct to evaluate the shell parameters $#.

```
#!/bin/sh
#emp3a.sh : Using Test ,$0 and $# in an if—elif –if construct.
 if  test $#  -eq 0; then
        echo "Usage:$0 pattern file">/dev/tty
 elif  test $# -eq 2 ; then
         grep "$1"  $2 || echo "$1 not found in $2>/dev/tty
 else
        echo "You did'nt enter two arguments">dev/tty
  fi
```

File Tests

- The Test command has several options for checking the status of a file.

These are shown in Fig below. Using these we can find out whether the specified file is an ordinary file, directory or whether it grants read,write or execute permissions,so on and so forth.

Option	Meaning
-s file	True if file exists and has a size > 0
-f file	True if file exists and is not a directory
-d file	True if file exists and is a directory File
-c file	True if File exists and is character special File
-b file	True if File exists and is a block special file
- r file	True if File Exists and you have a read permission
-w file	True if File Exists and you have write permissions
-x file	True if File Exists and you have execute Permission.

Sample Shell Script on File Tests

- echo “Enter Filename”
- read fname
- if [-f $fname]
- Then
- echo “You Indeed entered a Filename”
- Else
- echo “What you entered is not a Filename”
- fi.

String Comparision

- test can be used to compare Strings with yet another set of operators.

Equality is performed with = and inequality with the C-type operator !=.

Thus [!-z $string] negates [-z string]

String test used by Test.

Test	True if
s1=s2	String s1 = s2
s1!=s2	String s1 is not equal to s2.
-n stg	String stg is not a null string
-z stg	String stg is a null string.
Stg	String stg is assigned and not null.
s1==s2	String s1=s2 [Korn and Bash only]

String Comparision

- #SS19
- #Usage Ss19
- Str1="Good"
- Str2="Bad"
- Str3=
- [$str1 = $str2]
- echo $?
- [$str1 != $str2]
- echo $?
- [-n $str1]
- echo $?
- [-z $str3]
- echo $?
- ["$str3"]
- echo $?
- The O/p of the above program is 1 0 0 0 test:Argument expected.

String Comparision

- Why did we get the error test:Argument expected ? Because str3 is a null string and as we know a null string is ignored by the shell.
- Hence tests gets to work on [-z] hence the error.
- This Situation can be avoided as shown in the fourth test.This time we have enclosed str3 with " ".
- Therefore this time the test command gets to work with [-z " "] which returns a 0 since "" indeed represents a zero-length string.
- Observe that while carrying out the equality test there is a space on either side of '=' .This is necessary.Devoid of spaces it would become a simple assignment.
-

String Comparision.

- ##SS20
- str1="Good Morning"
- str2="Good Bye"
- [$str1 = $str2]
- echo $?
- Here is the output of the program test:unknown operator morning.
- Since two word strings are being assigned to variables str1 and str2 ,we have taken care to enclose the strings within a pair of double quotes .Still

the test failed.The test command went to work with [Good Morning = Good Bye]. Naturally morning was treated as an operator ,hence the error.

CASE CONTROL CHARACTER

- Another way of controlling the sequence of execution is using the case control structure. Though if-else constructs can be nested,an abundance of conditions may make tracing the control in a program difficult.
- The general form of the case control instruction is given below.
- case value in
- choice 1)do this *)do this
- and this ; ;
- ;; ; ;
- choice 2)do this esac
- and this
- ;;
- choice 3)do this

CASE CONTROL CHARACTER

- The “do this” “and” “and this “ lines in the above form represent any valid unix command. Also choice1,choice2 and choice 3 are labels which identify the potential choices of action.

- Firstly the expression following the case keyword is evaluated; the value that it yields is then matched ,one by one against the potential choices [choice1,choice2 and choice3 in the above form.]
- When a match is found the shell executes all commands in that case upto ;;

  This pair of semicolons placed at the end of each choice are necessary.
- They mark the end of statements within that choice.On encountering semicolons the control is transferred to esac,the keyword denoting the end of case .If we omit the semicolons an error results.

CASE CONTROL STRUCTURE

- Observe that the last case ‘*’) this represents the default clause of the case control instruction.
- It gets executed when all other cases fail.
- We can use Shell’s pattern matching abilities in the choice labels.In fact we have used one example already.When we used * to match any pattern.
- If we have no default case, and no case is satisfied then nothing is done and the program simply moves on to whatever comes after the case –esac statement.

Expr command in Shell Programming

- You may recall that all shell variables are string variables.If we are to carry out arithmetic operations on them we have to use the command expr which is capable of evaluating expression.The following program shows the various arithmetic operations that can be carried out using expr.
- a=20 b=10
- echo ‘expr $a + $b`
- echo `expr $a - $b`
- echo `expr $a  \* $b`
- echo ` expr $a / $b’
- echo `expr  $a % $b`
- On execution of this script we get the following output
- 30 100 200 2 0

Expr command in Shell Programming

- There are several things that should be noted in the above program.
- More than one assignment can be done in a single statement.Hence we initiated variables a and b in the same line.
- In addition to the normal addition,subtraction,multiplication and division the expr command can also perform a modular division operation.
- The Multiplication symbol must always be preceded by \.Otherwise the shell treats it as a wildcard character for all files in the current directory.
- Terms of the expression provided to expr must be separated by blanks.Thus the expression expr 10+20 is invalid.
- While evaluating an expression expr performs the various arithmetic operations according to the following priorities.
- /,* ,% - First Priority
- +,- Second priority

Expr Command in Shell Programming

- In case of tie between operations of same priority ,preference is given to the operator which occurs first.
- For Eg in the expression $a * $b + $c / $d, so $a * b would be performed before $c /$d.Since * appears prior to the / operator.
- expr is often used with command substitution to assign a variable.For example ,you can set a variable z to the sum of two numbers.
- $x=6 y=2; z=`expr $x + $y` $echo $z
- Perhaps the most common use of expr is in incrementing the value of a variable. $x=5 $x= `expr $x + 1` echo $x.

Loops in Shell Scripting

- There are three methods by way of which we can repeat a part of the program.They are
- Using a for statement .)Using a while statement .)Using an Until statement.
- The General form of while is as shown below
- while control command
- do
- command 1
- command 2
- done
- The statements within while loop would keep on getting executed till the exit status of the control command remains true.

Loops in Shell Scripting

- When the exit status of the control command turns out to be false ,the control passes to the first command that follows the body of the while loop.
- The Control command can be any valid unix command
- while [$# -le 5]
- while [-r $file -a -w $file]
- The statements within the loop may be a single command or a group of commands .In either case ,it is necessary to put them between do and done.
- Immediately following the while is a control command (the test command in our program).So long as the exit status of the control command is true,all commands within the body of the while loop keep getting executed rapidly.
-

Eg Program of While Loop

- #Calculation of Simple Interest for 3 sets of p,q,and r .
- Count=1
- while [$count –le 3]
- do
- echo “\n Enter values of p,n and r \c”
- read p r n
- si = `echo $p \* $n \* $r / 100 | bc `
- echo “Simple Interest= Rs $si”
- count = `expr $count + 1`
- done

For Looping with a List

- The Shell's for loop differs in structure from the one's used in other programming languages.Unlike while and until ,for does'nt test a condition but uses a list instead.
- for variable in list
- do
- commands Loop body
- done
- The loop body also uses the keywords do and done, but the additional parameters here are variable and list.
- Each whitespace – separated word in the list is assigned to variable in turn, and commands are executed until list is exhausted.

For Looping with a List

- Eg $for file in chap20 chap21 chap22 chap23
- do
- cp $file ${file}.bak
- echo $file copied to $file.book
- done
- chap20 copied to chap20.bak
- chap21 copied to chap21.bak
- chap22 copied to chap22.bak
- The list here comprises a series of character strings separated by whitespace.Each item in the list is assigned to the variable file.file first gets the value chap20 ,the chap21 and so on.
- The List can consists of practically any of the expressions that the shell understands and processes for

List from Command Substitution

- You can also use command substitution to create the list.The following for command line picks up its list from the file clist:
- $for file in `cat clist`
- This method is most suitable when the list is large and you don't consider it practicable to specify its contents individually.

- From Command Line Arguments
- #!/bin/sh
- for pattern in "$@"
- do
- grep "$pattern" emp.lst ||echo "$pattern not found"
- done

Setting Values of Positional Parameters

- There is one more way to assign values to positional parameters using the set command.
- $set BMS Computer science Department
- $echo $1 $2 $3 $4
- BMS Computer Science Students
- On giving another set command,the old values of $1,$2 etc are discarded and the new values get collected.
- $set Do you want credit or results
- $echo $1 $2 $3 $4 $5 $6

Setting Values of Positional Parameters

- Suppose we have typed the line ‘Give luck a little time and it will surely change` in a file called lucky.
- We can make set the values to be assigned to positional parameters from this file by saying
- $set `cat lucky`
- $ echo $1 $2 $3 $4 $5
- Give luck a little time
- The Date command by default displays the current date and time in the following format:
- Fri March 27 11:30 45 2015

Setting Values of Positional Parameters

- To display the Information in any other order say this:
- Fri 27 March 2015
- We can use the set command as shown as in the below program.
- set `date`
- echo $1 $3 $2 $5
- All we have done is juggled the position of the various positional parameters.To find out how many positional parameters were set either by set command or by command line arguments there is a special parameter $#.

Using Shift on Positional Parameters

- $Set India is the Seventh Largest Country in the World.India is the Second Largest Country in Asia.
- echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11
- India is the Seventh Largest Country in the world. India0 India1
- Observe that the last two words in the output.These Occurred in the output because at a time we can access only 9 positional parametrs.
- When we tried to refer to $10 it was interpreted by the shell as if you wanted
- To output the value of $1 and a 0.
- Hence we got India0 in the output.Same in the case of $11
-

Using Shift on Positional Parameters

- Does that mean the words following the ninth word have been lost?
- No They are very much there ,safe with the shell.But to reach them we must do the following.
  - Shift 8
  - echo $1 $2 $3 $4 $5 $6 $7 $8 $9
  - World.India is the second Largest country in Asia
  - The first 8 words have been shifted out. Each word vacated a position for the one on its right with the first word getting lost in the bargain.
  - This occurred 8 times hence we find last 9 words in $1 through $9.

Debugging a Script

- We would like to debug the program by tracing the flow of control,examining the values of variables checking whether the varianle,filename and command substitution is being done properly or not etc.
- To achieve this we have to simply add the following statement at the beginning of the script
- set –vx
- Here v ensures that each line in the shell script is displayed before it gets executed and x ensures that the command along with the argument

  values that it may have is also displayed before execution.

Debugging a Script

- Note that by setting the –v option the line from the script which is about to get executed is displayed whereas the lines which are preceded by + sign have come courtesy the –x option.

  Also note that in the lines preceded by the + sign the variables have been Substituted by their actual values.

Here Document

- We'll now attempt something that has far reaching consequences .Recall that we used an Interactive program emp1.sh by keying in two parameters.
- We can make the scripts work non-Interactively by supplying the Inputs through a here document.
- $emp1.sh << END
- >director
- >emp.lst
- END
- Enter the pattern to be searched:Enter the file to be used :Searching for
- director from file emp.lst

Here Document

- Note: The Contents of here Document are Interpreted and processed by the shell before they are fed as Input to a command.
- This means you can use command substitution and variables in its input.
- You can’t do that with normal standard Input.
- Many command require Input from the user .Often it’s the same input that is keyed in response to a series of questions posed by the command.
- We can instruct the inputs through a here Document.
- If you write a script that uses the read statement and which often assumes a predefined set of replies ,you can make the script behave non-interactively

by supplying its Input from a here document.

trap Command

- While using any of these methods we are sending certain signals to Unix.But sometimes the requirement is such that we want some process to ignore these signals.
- We want some process to carry out some other job on receiving signals.
- Unix Provides a command called trap to achieve this.Using this command,we can see to it that the program terminating signals can be persuaded to ignore the task that they were initially instructed to do.

Trap command

-

SIGNAL	SIGNAL NUMBER
Exit	0
ctrl +d	1
Del	2
Ctrl \	3
Sure Kill	9
Kill	15