Shell Script

What is Shell?

- A shell is an environment in which we can run our commands, programs, and shell scripts.

- There are different flavors of shells, just as there are different flavors of operating systems.

- Each flavor of shell has its own set of recognized commands and functions.

Shell Types

- In UNIX there are two major types of shells:
- **The Bourne shell.**
  - If you are using a Bourne-type shell, the default prompt is the $ character.
    - Bourne shell (/bin/sh)
    - Korn shell (/bin/ksh)
    - Bourne again Shell (/bin/bash)

- **The C shell.**
  - If you are using a C-type shell, the default prompt is the % character.

Shell Scripts

- The basic concept of a shell script is execute a list of commands, which are listed in the order.

  – There are conditional tests, such as value A is greater than value B,
  – loops allowing us to go through massive amounts of data,
  – files to read and store data,
  – and variables to read and store data,
  – and the script may include functions.

-- Note all the shell scripts would have **.sh** extension.

Shebang or Not

- Before you add anything else to your script, you need to alert the system that a shell script is being started.

- This is done using the shebang construct.
- For example −

  Bourne Shell #!/bin/sh

  Bash Shell #!/bin/bash

- -- If a script doesn't contain shebang, the commands are executed using your shell

  $ echo $SHELL

  $ echo $0
- -- Different shells have slightly varying syntax.

Shell Comments

- You can put your comments in your script as follows −
    - *#!/bin/bash*
    - *# Author : Rns*
    - *# Script follows here:*
    - *pwd*
    - *ls*
- Now you save the above content and make this script executable as follows −
    - *$chmod +x script.sh*
- Now you have your shell script ready to be executed as follows −
- *$./script.sh*
- **Note:** To execute any program, you would execute using **./program_name.sh**

Using Shell Variables

- A variable is a string to which we assign a value.
- Storage locations that have a name
- The value assigned could be a number, text, filename, device, or any other type of data.
- Variables are case sensitive
- By convention variables are uppercase

- Syntax:

  VARIABLE_NAME="VALUE"

- For example:
  - NAME=“Rise N Shine“

- The shell enables you to store any value you want in a variable. For example −
  - VAR1=“Rise N Shine"
  - VAR2=100

Variable Names

- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

- The following examples are valid variable names
    - _ALI
    - TOKEN_A
    - VAR_1
    - VAR_2
- Following are the examples of invalid variable names −
    - 2_VAR
    - -VARIABLE
    - VAR1-VAR2

Accessing Values

- To access the value stored in a variable, prefix its name with the dollar sign ($) −
- For example, following script would access the value of defined variable NAME and would print it on STDOUT −
    - *#!/bin/sh*
    - *NAME=“Rise N Shine"*
    - *echo $NAME*
- This would produce following value −
    - Rise N Shine

Lab Exercise

Read-only Variables

- The shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

- For example, following script would give error while trying to change the value of NAME −
  - *#!/bin/sh*
  - *NAME=$1*
  - *readonly NAME*
  - *NAME="RNS"*

- This would produce following result −
  - /bin/sh: NAME: This variable is read only.

Variable Types

- When a shell is running, three main types of variables are present −

- **Variables** − A variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.

- **Environment Variables** − An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

- **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Special Variables

- The following shows a number of special variables that you can use in your shell scripts

- **$0:**The filename of the current script.

- **$n** These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).

- **$#**The number of arguments supplied to a script.

Special Variables

- **$***All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.

- **$@**All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.

- **$?**The exit status of the last command executed.

- **$$**The process number of the current shell. For shell scripts, this is the process ID under which they are executing.

- **$!**The process number of the last background command.

Special Parameters $* and $@

- There are special parameters that allow accessing all of the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".

- the "$*" special parameter takes the entire list as one argument with spaces between

- the "$@" special parameter takes the entire list and separates it into separate arguments.

Shell Decision Making

- Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will explain following two decision making statements −

  – The **if...else** statements

The if...else statements:

- If else statements are useful decision making statements which can be used to select an option from a given set of options.
- Unix Shell supports following forms of if..else statement −
  - if...fi statement
  - if...else...fi statement
  - if...elif...else...fi statement
- Most of the if statements check relations using relational operators discussed in previous chapter.

The if...fi statement

- The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

- Syntax:
  - if [expression]
    - –then Statement(s) to be executed if expression is true
  - fi

Lab Exercise -

The if...else...fi statement

- The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.
- Syntax:
  - if [expression]
  - then
    - Statement(s) to be executed if expression is true
  - else
    - Statement(s) to be executed if expression is not true
  - fi

The if...elif...fi statement

- The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.
- Syntax:

```
if [ expression 1 ]
then
    Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
    Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
    Statement(s) to be executed if expression 3 is true
else
    Statement(s) to be executed if no expression is true
fi
```

Shell Basic Operators

- There are following operators which we are going to discuss −
  - Arithmetic Operators.
  - Relational Operators.
  - Boolean Operators.
  - String Operators.
  - File Test Operators.
- The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs the must simpler program **expr**.
- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
  - $ echo `expr $a + $b`
- Complete expression should be enclosed between ``, called inverted commas.

Arithmetic Operators

-- Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
+	Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

-- Lab Exercise --

Relational Operators:

- Assume variable a holds 10 and variable b holds 20 then −

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	[$a -le $b] is true.

Lab Exercise --

Boolean Operators

- Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

Lab Exercise --

String Operators

- Assume variable a holds "abc" and variable b holds "efg" then −

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	[-z $a] is not false.
str	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

Lab Exercise --

File Test Operators

- Assume a variable **file** holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on −

Operator	Description	Example
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-c $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.

File Test Operators

- Lab Exercise --

-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.

The case...esac Statement

- You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

- Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

- There is only one form of case...esac statement

The case...esac Statement

- Syntax
- The basic syntax of the case...esac statement is to give an expression to evaluate and several different statements to execute based on the value of the expression.
- case word in
  - pattern1)
    - Statement(s) to be executed if pattern1 matches
    - ;;
  - pattern2)
    - Statement(s) to be executed if pattern2 matches
    - ;;
  - pattern3)
    - Statement(s) to be executed if pattern3 matches
    - ;;
  - esac
- Lab Exercise –

Shell Loop Types

- Loops are a powerful programming tool that enable you to execute a set of commands repeatedly. In this tutorial, you would examine the following types of loops available to shell programmers −

  – The for loop
  – The while loop

The for Loop

- The for loop operate on lists of items. It repeats a set of commands for every item in a list.

- Syntax
  - *for var in word1 word2 ... wordN*
  - *do*
    - *Statement(s) to be executed for every word.*
  - *done*

The while Loop

- The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

- Syntax
  - while command
  - do
    - Statement(s) to be executed if command is true
  - done

Nesting Loops

- Nesting while Loops
- It is possible to use a while loop as part of the body of another while loop.
- Syntax
  - while command1 ; # this is loop1, the outer loop
  - do
    - Statement(s) to be executed if command1 is true
    - while command2 ; # this is loop2, the inner loop
    - do
      - Statement(s) to be executed if command2 is true
    - done
    - Statement(s) to be executed if command1 is true
  - done

Shell Loop Control

- Sometimes you need to stop a loop or skip iterations of the loop.
- The following two statements used to control shell loops −
  - The **break** statement
  - The **continue** statement

The infinite Loop

- All the loops have a limited life and they come out once the condition is false or true depending on the loop.

- A loop may continue forever due to required condition is not met. A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called infinite loops.

The break statement

- The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.
- Syntax
- The following **break** statement would be used to come out of a loop −
  - Break
- The break command can also be used to exit from a nested loop using this format −
  - break n
- Here **n** specifies the nth enclosing loop to exit from.

The continue statement

- The **continue** statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

- This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

- Syntax
  - *continue*
- Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.
  - *continue n*
- Here n specifies the nth enclosing loop to continue from.

Shell Functions

- Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

- Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

- To declare a function, simply use the following syntax −

```
function_name () {
    list of commands
}
```

- The name of the function is function_name, and that's what you will use to call it from elsewhere in your scripts.

- The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.