Experiment No : 4

Aim : Shell scripting: study bash syntax, environment variables, variables, control constructs such as if, for and while, aliases and functions, accessing command line arguments passed to shell scripts. Study of startup scripts, login and logout scripts, familiarity with systemd and system 5 init scripts is expected.

Shell Scripting

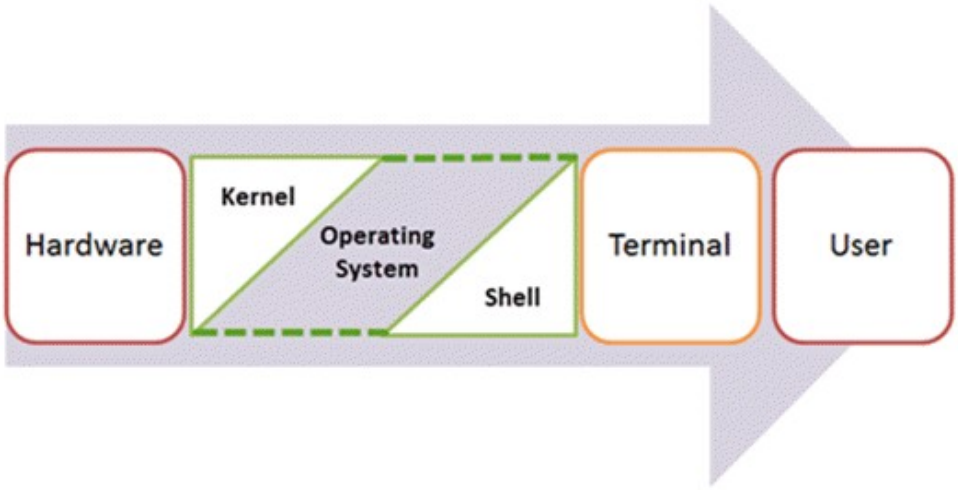
Shell Scripting is an open-source computer program designed to be run by the Unix/Linux shell. Shell Scripting is a program to write a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script that can be stored and executed anytime which, reduces programming efforts.

Shell

Shell is a UNIX term for an interface between a user and an operating system service. Shell provides users with an interface and accepts human-readable commands into the system and executes those commands which can run automatically and give the program's output in a shell script.

An Operating is made of many components, but its two prime components are -

- Kernel
- Shell



A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one.
A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.
When you run the terminal, the Shell issues **a command prompt (usually $),** where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.
The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name **Shell**.

Types of Shell

There are two main shells in Linux:

1. The **Bourne Shell**: The prompt for this shell is $ and its derivatives are listed below:

- POSIX shell also is known as sh
- Korn Shell also knew as sh
- **B**ourne **A**gain **SH**ell also knew as bash (most popular)

2. The C shell: The prompt for this shell is %, and its subcategories are:

- C shell also is known as csh
- Tops C shell also is known as tcsh

Why do we need shell scripts

There are many reasons to write shell scripts –

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups
- System monitoring
- Adding new functionality to the shell etc.

Advantages of shell scripts

•The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax

•Writing shell scripts are much quicker

•Quick start

•Interactive debugging etc.

Disadvantages of shell scripts

•Prone to costly errors, a single mistake can change the command which might be harmful

•Slow execution speed

•Design flaws within the language syntax or implementation

•Not well suited for large and complex task

•Provide minimal data structure unlike other scripting languages. etc

Steps in creating a Shell Script:

1. Create a file using a **vi** editor(or any other editor). Name script file with **extension .sh**
2. **Start** the script with **#! /bin/sh**
3. Write some code.
4. Save the script file as filename.sh
5. For **executing** the script type **bash filename**

Shell Variables

As discussed earlier, Variables store data in the form of characters and numbers. Similarly, Shell variables are used to store information and they can by the shell only.
For example, the following creates a shell variable and then prints it:

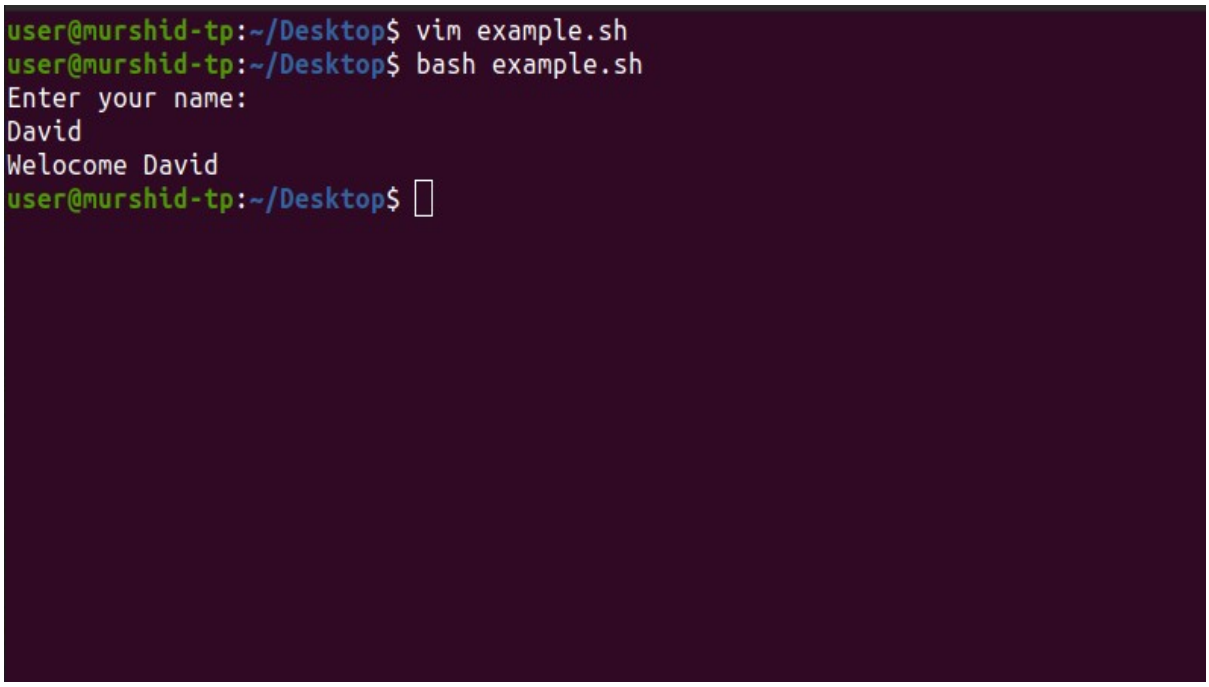
```
variable ="Hello"
echo $variable
```

Relational Operators

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

Example:

vim example.sh



Conditional Statements: There are total 5 conditional statements which can be used in bash programming

1.if statement

2.if-else statement

3.if..elif..else..fi statement (Else If ladder)

4.if..then..else..if..then..fi..fi..(Nested if)

5.switch statement

Their description with syntax is as follows:

if statement

This block will process if specified condition is true.

Syntax:

```
if [ expression ]
then
   statement
fi
```

if-else statement

If specified condition is not true in if part then else part will be execute.

Syntax

```
if [ expression ]
then
   statement1
else
   statement2
fi
```

if..elif..else..fi statement (Else If ladder)

To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part.

Syntax

```
if [ expression1 ]
then
   statement1
   statement2
   .
   .
```

```
elif [ expression2 ]
then
  statement3
  statement4
  .
  .
else
  statement5
fi
```

if..then..else..if..then..fi..fi..(Nested if)

Nested if-else block can be used when, one condition is satisfies then it again checks another condition. In the syntax, if expression1 is false then it processes else part, and again expression2 will be check.

Syntax:

```
if [ expression1 ]
then
  statement1
  statement2
  .
else
  if [ expression2 ]
  then
    statement3
    .
  fi
fi
```

switch statement

case statement works as a switch statement if specified value match with the pattern then it will execute a block of that particular pattern

When a match is found all of the associated statements until the double semicolon (;;) is executed.

A case will be terminated when the last command is executed.

If there is no match, the exit status of the case is zero.

Syntax:

```
case  in
  Pattern 1) Statement 1;;
  Pattern n) Statement n;;
esac
```

Example :

```
#!/bin/sh

echo "Enter a number:"
read n
if [ $((n % 2)) -eq 0 ]
then
        echo "No is even"
else
        echo "No is odd"
fi
~
~
~
~
~
~
~
~
~
~
```

```
user@murshid-tp:~/Desktop$ vim example.sh
user@murshid-tp:~/Desktop$ bash example.sh
Enter a number:
34
No is even
user@murshid-tp:~/Desktop$ bash example.sh
Enter a number:
23
No is odd
user@murshid-tp:~/Desktop$
```

Looping Statements in Shell Scripting: There are total 3 looping statements which can be used in bash programming

1.while statement

2.for statement

3.until statement

Their descriptions and syntax are as follows:

while statement

Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated

Syntax

```
while command
do
   Statement to be executed
done
```

for statement

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

Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Syntax

```
for var in word1 word2 ...wordn

do
   Statement to be executed
done
```

until statement

The until loop is executed as many as times the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

Syntax

until command
do
  Statement to be executed until command is true
done

Example :

```
#!/bin/sh

for i in 0 1 2 3 4 5 6 7
do
        echo $i
done
~
~
~
~
~
~
~
~
~
~
~
```

```
user@murshid-tp:~/Desktop$ vim example.sh
user@murshid-tp:~/Desktop$ bash example.sh
0
1
2
3
4
5
6
7
user@murshid-tp:~/Desktop$
```

Functions in Shell script

Although aliases are quick and easy to implement, they are quite limited in their scope. You'll find as you're trying to chain commands together that you can't access arguments given at runtime very well, among other things. Aliases can also be quite slow at times because they are read after all functions.

There is an alternative to aliases that is more robust and can help you bridge the gap between bash aliases and full shell scripts. These are called shell functions. They work in almost the same way as aliases but are more programmatic and accept input in a standard way.

We won't go into extensive detail here, because these can be used in so many complex situations and bash is an entire scripting language, but we'll go over some basic examples.

For starters, there are two basic ways to declare a bash syntax. The first uses the function command and looks something like this:

```
function function_name {
   command1
   <^>command2</^>
}
```

The other syntax uses a set of parentheses which is more "C-like":

```
function_name () {
   command1
   command2
}
```

We can compress this second form into one line and separate the commands with semicolons. A semicolon *must* come after the last command too:

```
function_name () { command1; command2; }
```

Example

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World"
}
```