Experiment No.4

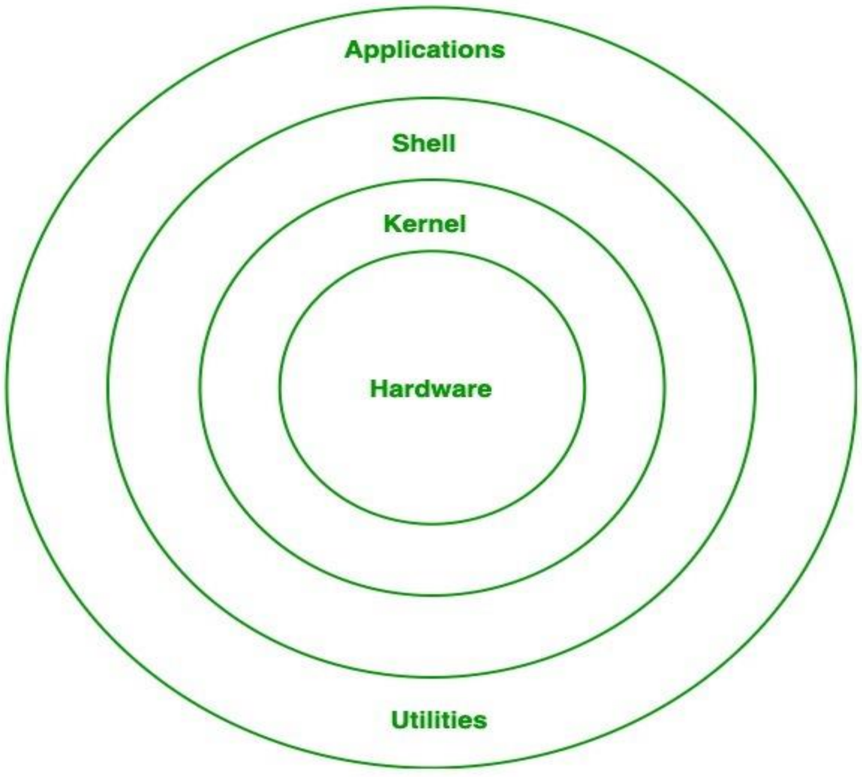
Aim

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.

Result

Shell Scripting

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.



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.

Shell is broadly classified into two categories –

- Command Line Shell
- Graphical shell

Command Line Shell : Shell can be accessed by user using a command line interface. A special program called Terminal in linux/macOS or Command Prompt in Windows OS is provided to type in the human readable commands such as “cat”, “ls” etc. and then it is being execute.

Graphical Shells : Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions.

Shell Prompt

The prompt, $, which is called the command prompt, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press Enter. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Shell Types

In Unix, there are two major types of shells −

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

The Bourne Shell has the following subcategories −

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow −

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

Bash

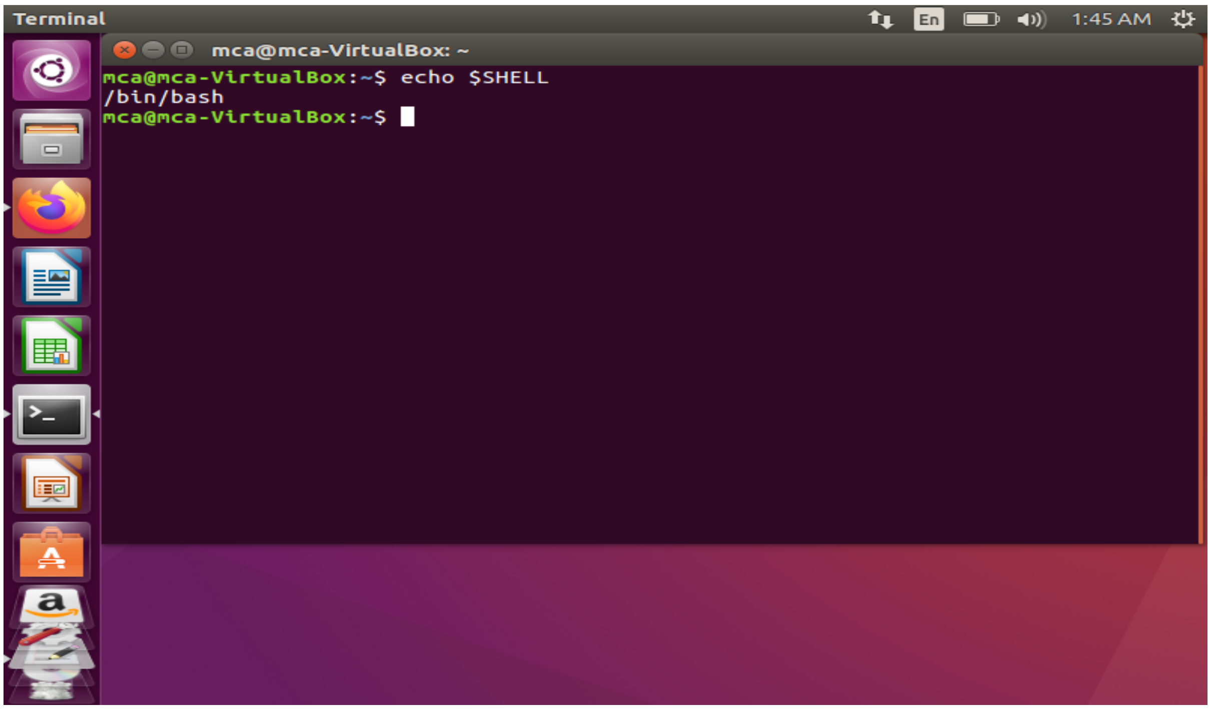
BASH is an acronym for Bourne Again Shell. Bash is a shell program written by Brian Fox as an upgraded version of Bourne Shell program '**sh**'. It is an open-source GNU project. It was released in 1989 as one of the most popular shell distribution of GNU/Linux operating systems. It provides functional improvements over Bourne Shell for both programming and interactive uses. It includes command line editing, key bindings, command history with unlimited size, etc.

In basic terms, Bash is a command line interpreter that typically runs in a text window where user can interpret commands to carry out various actions. The combination of these commands as a series within a file is known as a Shell Script. Bash can read and execute the commands from a Shell Script.Bash is the default login shell for most Linux distributions and Apple's mac OS. It is also accessible for Windows 10 with a version and default user shell in Solaris 11.

Example Script

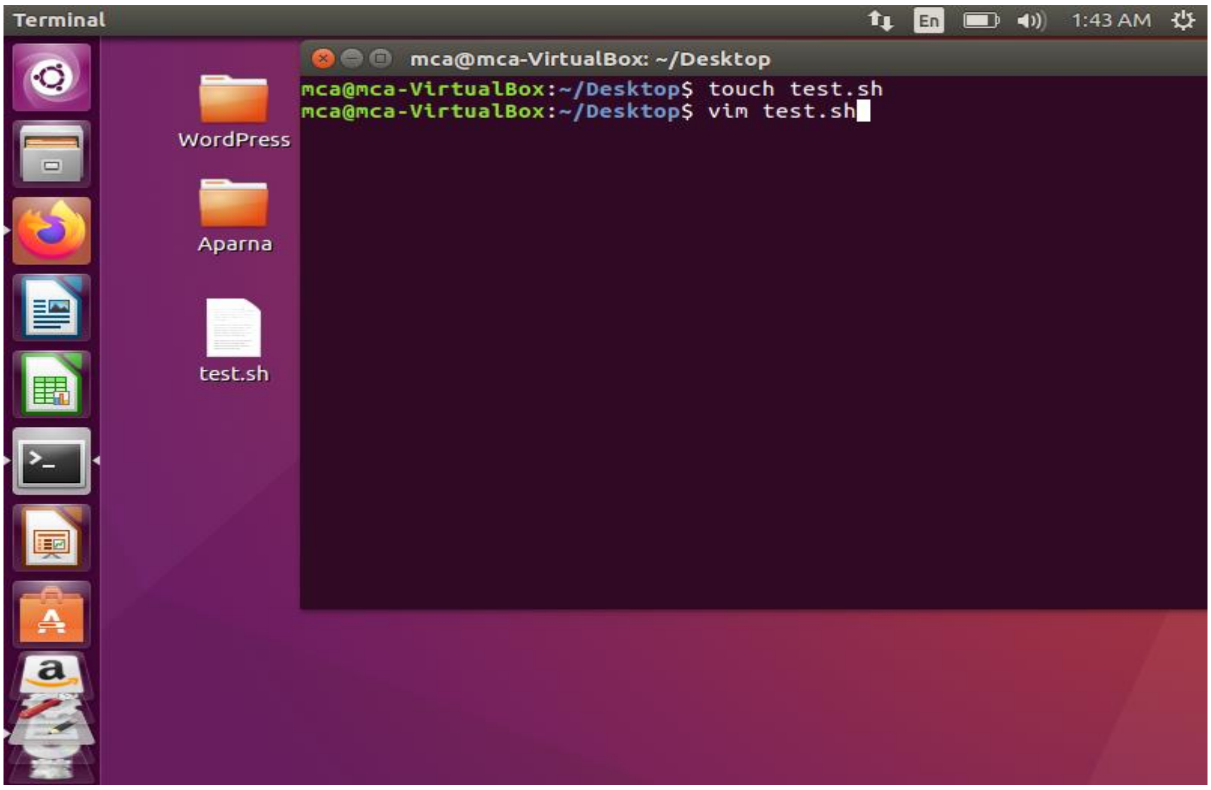
In a Bash Shell Script First you need to find out where is your bash interpreter located. Enter the following into your command line:

```
echo $shell
```



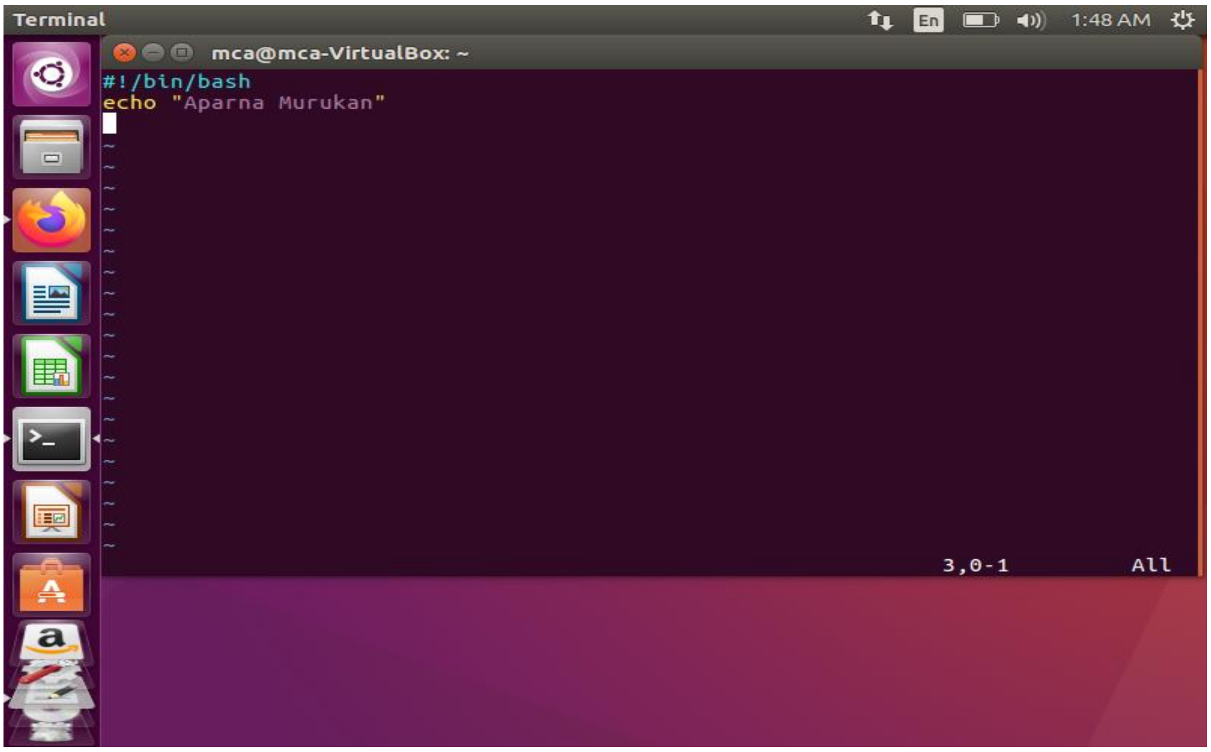
```
mca@mca-VirtualBox:~$ echo $SHELL
/bin/bash
mca@mca-VirtualBox:~$
```

- Open up vim text editor and create file called test.sh



```
mca@mca-VirtualBox:~/Desktop$ touch test.sh
mca@mca-VirtualBox:~/Desktop$ vim test.sh
```

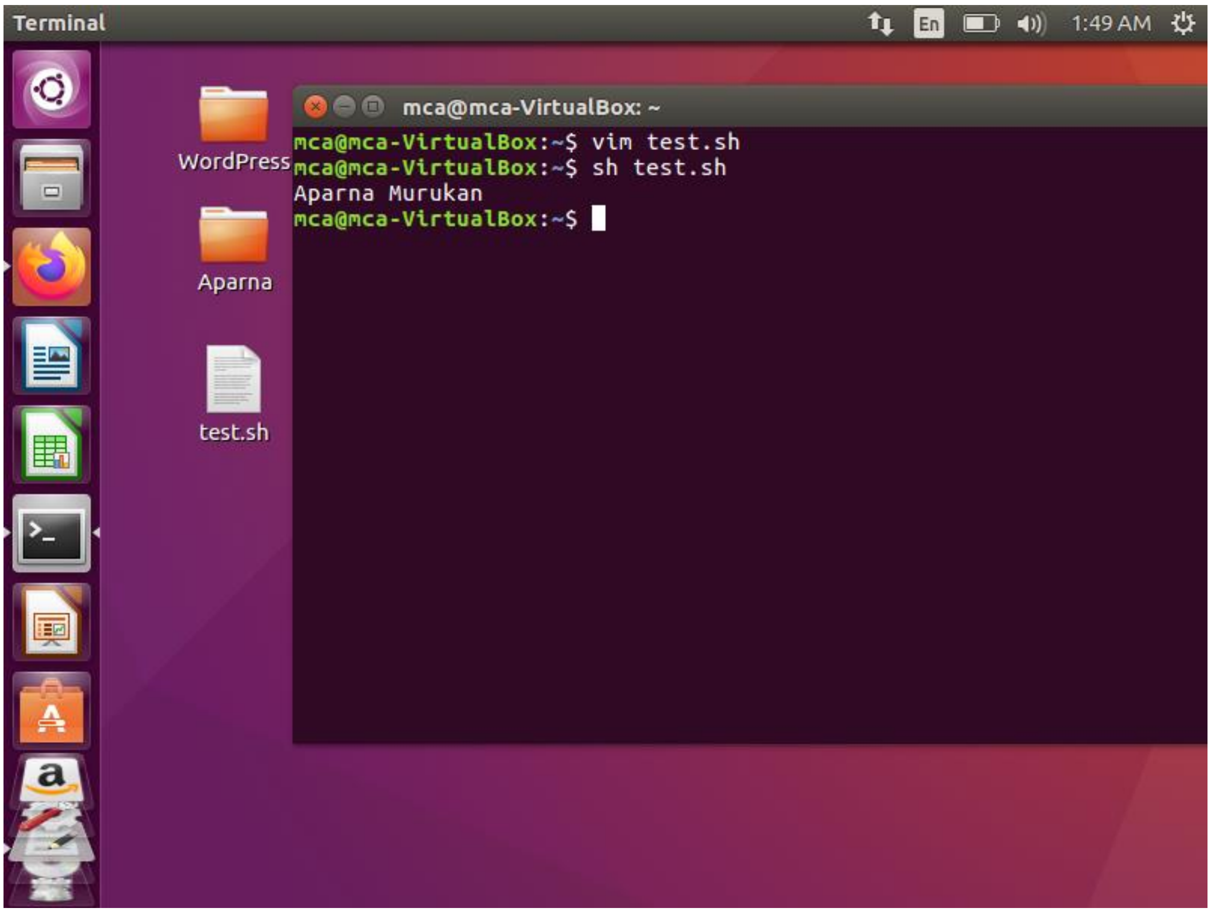
- Insert the following lines to a file:



```
#!/bin/bash
echo "Aparna Murukan"

~
~
~
                                                       3,0-1          All
```

- Execute the bash script:



```
mca@mca-VirtualBox:~$ vim test.sh
mca@mca-VirtualBox:~$ sh test.sh
Aparna Murukan
mca@mca-VirtualBox:~$ 
```

Shell Variables

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

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 (_). By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names −

_ALI
TOKEN_A
VAR_1
VAR_2

Defining Variables

Variables are defined as follows −

variable_name=variable_value

For example −

NAME="Zara Ali"

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (**$**) –

For example, the following script will access the value of defined variable NAME and print it on STDOUT −

```
#!/bin/sh
```

```
NAME="Zara Ali"
echo $NAME
```

The above script will produce the following value −

```
Zara Ali
```

Read-only Variables

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, the following script generates an error while trying to change the value of NAME −

```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

The above script will generate the following result −

```
/bin/sh: NAME: This variable is read only.
```

Variable Types

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

- Local Variables − A local 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 the command prompt.
- Environment Variables − An environment variable 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

Conditional Statements in Shell Script

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
```

Looping Statements in Shell Script

There are total 3 looping statements which can be used in bash programming

1. while statement
2. for statement
3. until statement

To alter the flow of loop statements, two commands are used they are,

1. break
2. continue

Their descriptions and syntax are as follows:

- <u>while statement</u> : 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
  ```

- <u>for statement</u> : 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
  ```

- <u>until statement: </u>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
  ```

Shell functions

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

<u>Creating Functions</u>

To declare a function, simply use the following syntax −

```
function_name () {

   list of commands

}
```

Example

Following example shows the use of function −

```
#!/bin/sh

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

# Invoke your function
Hello
```

Upon execution, you will receive the following output −

```
$./test.sh
Hello World
```

Pass Parameters to a Function

You can define a function that will accept parameters while calling the function. These parameters would be represented by **$1**, **$2** and so on.

Following is an example where we pass two parameters *Zara* and *Ali* and then we capture and print these parameters in the function.

```
#!/bin/sh

# Define your function here
Hello () {
  echo "Hello World $1 $2"
}

# Invoke your function
Hello Zara Ali
```

Upon execution, you will receive the following result −

```
$./test.sh

Hello World Zara Ali
```

Returning Values from Functions

If you execute an **exit** command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the return command whose syntax is as follows −

```
return code
```

Here code can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example

Following function returns a value 10 −

```
#!/bin/sh

# Define your function here
Hello () {
  echo "Hello World $1 $2"
  return 10
}
```

```
# Invoke your function
Hello Zara Ali

# Capture value returnd by last command
ret=$?

echo "Return value is $ret"
```

Upon execution, you will receive the following result −

```
$./test.sh

Hello World Zara Ali

Return value is 10
```

Nested Functions

One of the more interesting features of functions is that they can call themselves and also other functions. A function that calls itself is known as a *recursive function*.

Following example demonstrates nesting of two functions −

```
#!/bin/sh

# Calling one function from another
number_one () {
  echo "This is the first function speaking..."
  number_two
}

number_two () {
  echo "This is now the second function speaking..."
}

# Calling function one.
number_one
```

Upon execution, you will receive the following result −

```
This is the first function speaking...

This is now the second function speaking...
```

Function Call from Prompt

You can put definitions for commonly used functions inside your *.profile*. These definitions will be available whenever you log in and you can use them at the command prompt.Alternatively, you can group the definitions in a file, say *test.sh*, and then execute the file in the current shell by typing −

```
$. test.sh
```

This has the effect of causing functions defined inside *test.sh* to be read and defined to the current shell as follows −

```
$ number_one

This is the first function speaking...

This is now the second function speaking...

$
```

To remove the definition of a function from the shell, use the unset command with the .f option. This command is also used to remove the definition of a variable to the shell.

```
$ unset -f function_name
```