Birla Institute of Technology & Science, Pilani – K. K. Birla Goa campus
Computer Programming (CS F111)
Second Semester 2013-2014
Lab-03 (UNIX- Shell)

Objectives

1. Introduction to Unix Shell and Pattern Matching
2. Shell Script
3. Interactive Shell Scripts
4. Performing Arithmetic Operations
5. Variables in Shell
6. Decision making constructs in Shell
7. Exercises

---

1. What is Shell?

A program that interprets commands and allows a user to execute commands by typing them manually at a terminal, or automatically in programs called ***shell scripts.***
A shell is *not* an operating system. It is a way to interface with the operating system and execute commands.
Login shell is BASH = **B**ourne **A**gain **Sh**ell

Bash is a shell written as a free replacement to the standard Bourne Shell (/bin/sh) originally written by Steve Bourne for UNIX systems. It has all the features of the original Bourne Shell, plus additions that make it easier to program with and use from the command line. Since it is Free Software, it has been adopted as the default shell on most Linux systems.
Many commands accept arguments which are file names. For example:

```
ls -l main.c
```

prints information relating to the file *main.c*.
The shell provides a mechanism for generating a list of file names that match a pattern.
For example,

```
ls -l *.c
```

generates, as arguments to *ls,* all file names in the current directory that end in ***.c***.
The character * is a pattern that will match any string including the null string.
This mechanism is useful both to save typing and to select names according to some pattern.
Following are the special characters interpreted by the shell called as ***Wild cards***

Command	What does the command do?
*	Matches any number of characters including none
?	Matches a single character
[ijk]	Matches a single character either i , j or k
[!ijk]	Not i, j or k
[x-z]	At least a single character within this ASCII range
[!x-z]	Not a single character within this range

(i) * stands for zero or more characters.

Examples:

ls a	**(output is only a if a is a file, output is the content of a if a is a directory)**
ls ar*	**(all those that start with ar)**
ls *vict	**(all those files that end with vict)**
ls *[ijk]	**(all files having i ,j or k as the last character of the filename)**
ls *[b-r]	**(all files which have at least any character between b and r as the last character of the filename)**
ls *	**(all files)**

(ii) ? matches a single character

Examples:

ls a?t	matches all those files of three characters with **a** as the first and **t** as the third character and any character in between
ls ?oo	matches all three character files whose filename end with **oo**

What will the following commands give?

Command	**What does it do?**
ls ??i*	Matches any number of characters but definitely two characters before i followed by any number of characters
ls ?	Matches all the single character files
ls *?	Matches any file with at least 1 character
ls ?*	Matches any file with at least 1 character
ls “*”	Matches any file with * as the filename (exactly 1 character which is *)
ls “*”*	Matches all files starting with * as filename

1.1 Quoting

Characters that have a special meaning to the shell, such as **< > * ? | &,** are called *metacharacters*. Any character preceded by a \ is *quoted* and loses its special meaning, if any.

The \ is elided so that

echo \?

will echo a single **?** and

echo \\

will echo a single \
\ is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes.

For example,

echo xx'**'xx**

will echo

xx**xx**

The quoted string may not contain a single quote but may contain new lines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters.

2. Shell Scripts

The shell may be used to read and execute commands contained in a file.

As a simple case, assume that we have a scenario where commands who, ls and date are executed one after the other.
To write shell scripts open a new file in vi (say vi firstShellPgm)
Type all the commands that you want to get executed one after the other. i.e. who, ls and date
The file content will be now

```
echo "My First Shell Program"
who
ls -l
date
```

As stated earlier, the shell may be used to read and execute commands contained in a file.
To read and execute the file use the following in command line

sh file [args ...] ***e.g. sh firstShellPgm***

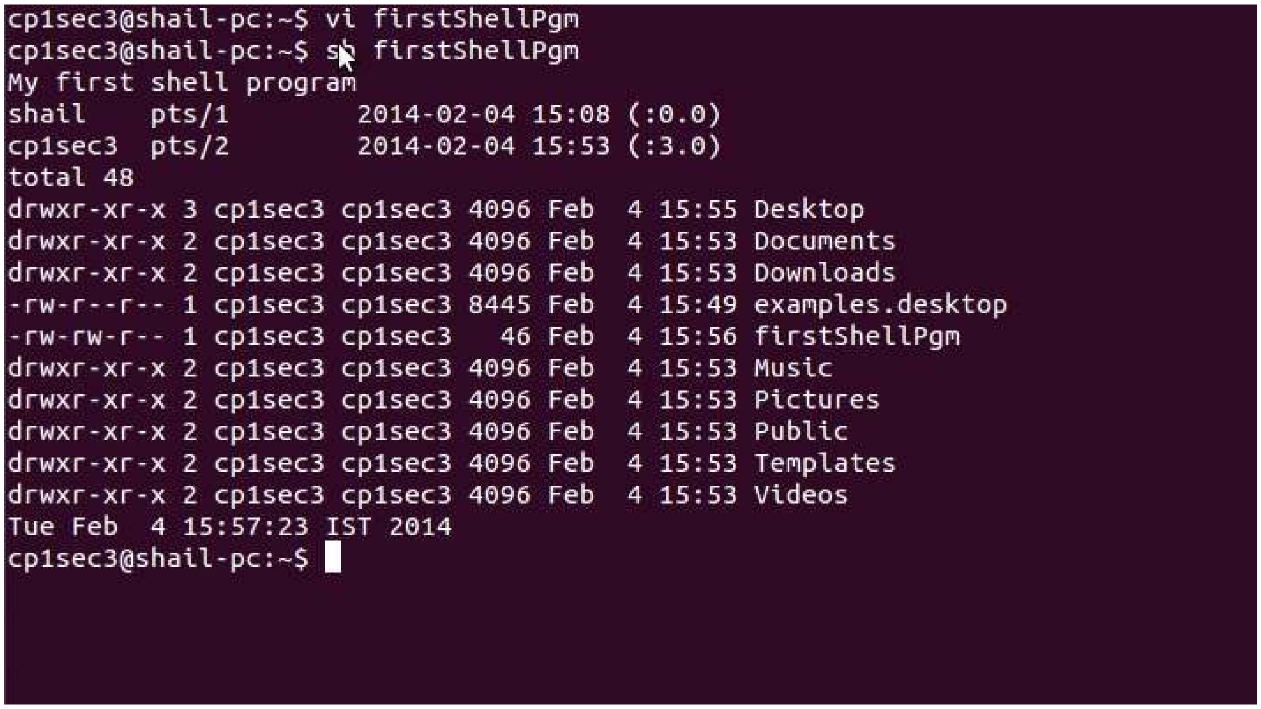
sh calls the shell to read commands from *file.* Such a file is called a *command procedure* or *shell procedure or shell script.*

Execute the program by typing

sh ***firstShellPgm***

This produces the output as shown in Figure 1.

Note: You need to give execute permission to file firstShellPgm
chmod +x firstShellPgm



```
cp1sec3@shail-pc:~$ vi firstShellPgm
cp1sec3@shail-pc:~$ sh firstShellPgm
My first shell program
shail    pts/1        2014-02-04 15:08 (:0.0)
cp1sec3  pts/2        2014-02-04 15:53 (:3.0)
total 48
drwxr-xr-x 3 cp1sec3 cp1sec3 4096 Feb  4 15:55 Desktop
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Documents
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Downloads
-rw-r--r-- 1 cp1sec3 cp1sec3 8445 Feb  4 15:49 examples.desktop
-rw-rw-r-- 1 cp1sec3 cp1sec3   46 Feb  4 15:56 firstShellPgm
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Music
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Pictures
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Public
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Templates
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Videos
Tue Feb  4 15:57:23 IST 2014
cp1sec3@shail-pc:~$ 
```

Figure 1: Output of the sh firstShellPgm

Arguments may be supplied with the call and are referred to in *file* using the positional parameters **$1, $2, ...**
For example, if the file ***secondShellPgm*** contains

```
echo My Second Shell Program with Parameter Passing
echo The parameter1 is $1 and Parameter 2 is $2
who|grep $1
ls -l $2*
date
```

then
sh *secondShellPgm* CP1SEC3 first

produces the result as shown in Figure 2.

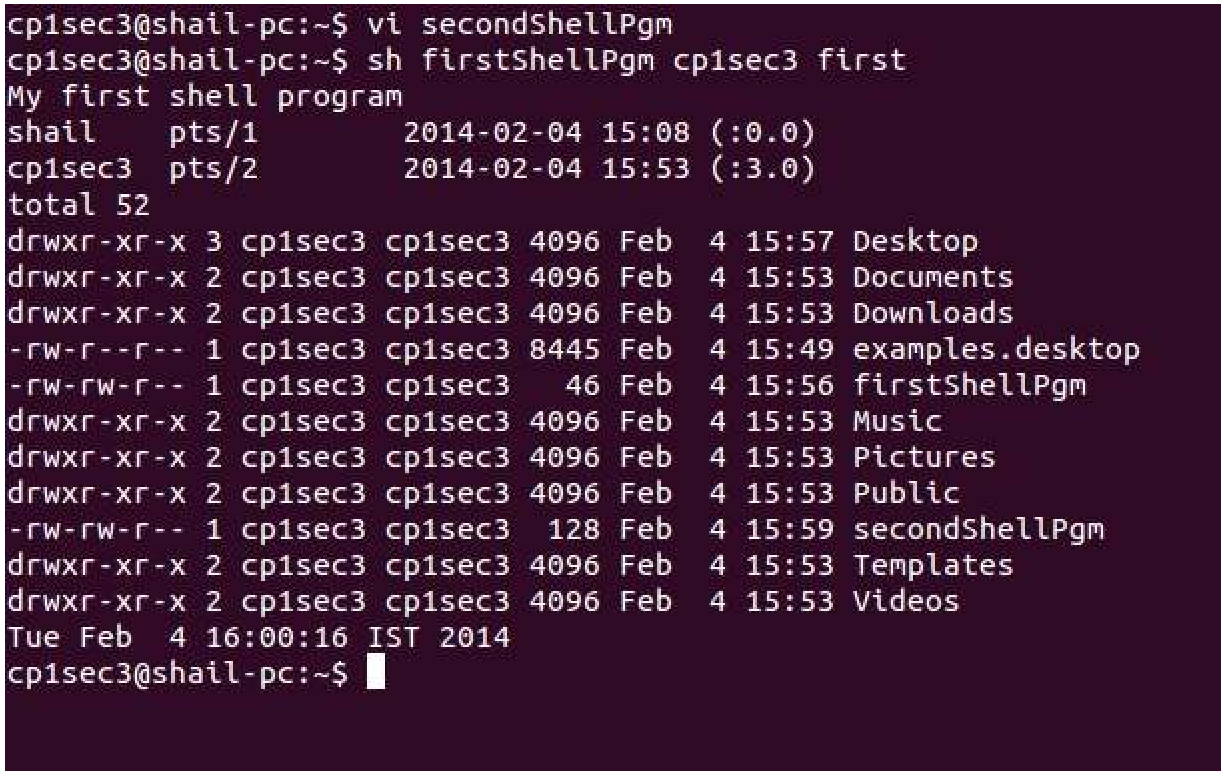

```
cp1sec3@shail-pc:~$ vi secondShellPgm
cp1sec3@shail-pc:~$ sh firstShellPgm cp1sec3 first
My first shell program
shail    pts/1        2014-02-04 15:08 (:0.0)
cp1sec3  pts/2        2014-02-04 15:53 (:3.0)
total 52
drwxr-xr-x 3 cp1sec3 cp1sec3 4096 Feb  4 15:57 Desktop
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Documents
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Downloads
-rw-r--r-- 1 cp1sec3 cp1sec3 8445 Feb  4 15:49 examples.desktop
-rw-rw-r-- 1 cp1sec3 cp1sec3   46 Feb  4 15:56 firstShellPgm
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Music
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Pictures
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Public
-rw-rw-r-- 1 cp1sec3 cp1sec3  128 Feb  4 15:59 secondShellPgm
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Templates
drwxr-xr-x 2 cp1sec3 cp1sec3 4096 Feb  4 15:53 Videos
Tue Feb  4 16:00:16 IST 2014
cp1sec3@shail-pc:~$ 
```

Figure 2: Output of the sh secondShellPgm CP1SEC3 first

When the file executes, the first argument (CP1SEC3 in the previous example) replaces $1, second argument replaces $2 …….
So

echo The parameter1 is $1 and Parameter 2 is $2

is equivalent to

echo The parameter1 is CP1SEC3 and Parameter 2 is first

and

```
who|grep $1
```

is equivalent to

```
who|grep CP1_SEC1
```

and

```
ls –l $2*
```

is equivalent to

```
ls –l first*
```

```
i.e. CP1SEC3 is assigned to variable 1 and first is assigned to
variable 2
```

Note: You need to give execute permission to file secondShellPgm
chmod +x secondShellPgm

3. Interactive Shell Scripts

Interaction with the computer is always through input and output operations. The ***read*** command allows the shell scripts to read input from the user. Let us write our first interactive shell script here. You may save it in the file ***first***.
The script is as follows

```
#this script asks user to input his name and then prints his name
echo What is your name \?
read name
echo Hello $name. Happy Programming.
```

Execute the script by typing

```
sh first
```

You must have noticed \ symbol before? This can be avoided if the message to be displayed is enclosed in quotes. This script causes the name entered by you is stored in the variable ***name***. Also, while using in echo statement, this variable name is attached with $ indicating that name is a variable, and extract the value stored in the variable ***name*** and use it in place of it.
The output is:

```
What is your name ?
john
Hello john. Happy Programming.
```

4. Performing Arithmetic operations

We start with an example here. Type the following code in the file named ***arithmetic***.

```
#this script performs different arithmetic operation on two numbers
a=$1
b=$2
echo `expr $a + $b'
echo `expr $a - $b'
echo `expr $a \* $b'         #multiplication
echo `expr $a / $b'
echo `expr $a % $b'          #modular division, returns reminder
```

Now execute the script by typing

```
sh  arithmetic 20  12
```

You will get answer as follows

```
32
8
240
1
8
```

Line 1 is commented which is used for improving readability of the script. This line will not be executed. Anything following # is a comment and will be only for improving readability.
Line 2 and 3 assigns some values [passed through argument 1 and argument 2] to variables a and b. In the line `echo 'expr $a + $b'` expr is the key value that signifies it as an arithmetic expression to be evaluated.
An expression like **$a * \\($b + $c \\) / $d** is a valid expression. This expression performs **a * (b+c) / d**. The enclosing punctuation (`' '`) in **`'expr $a + $b'`** is accent grave. It causes the expression to be evaluated. expr is only able to carry out the integer arithmetic.

NOTE: The symbol quoting expr can be found on key having Tilde (~) Below ESC key

5. Variables in Shell

To process our data/information, data must be kept in computer's RAM. RAM is divided into small locations, and each location had unique number called memory location/address, which is used to hold the data. Programmer can give a unique name to this memory location/address called memory variable or variable [It is a named storage location that may take different values, but only one at a time]. In Linux (Shell), there are two types of variable:
(1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
(2) **User defined variables (UDV)** - Created and maintained by user.

5.1 System Variables

Some of the important System variables are:

System Variable	Meaning
BASH=/bin/bash	Shell name
BASH_VERSION=3.2.9(1)-release	Shell version name
COLUMNS=95	Number of columns for the screen
HOME=/home/CP1SEC3	Home directory
LINES=24	Number of columns for the screen
LOGNAME=CP1SEC3	Log in name
OSTYPE=linux-gnu	OS type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Path settings
PS1=[\u@\h \W]\$	Prompt settings
PWD=/home/CP1SEC3/	Present working directory
SHELL=/bin/bash	Shell name
USERNAME=CP1SEC3	User name who is currently login to this PC

Caution: Do not modify System variable this can some time create problems.
You can see the current values of these variables by typing **echo $variable name**
Example: echo $HOME

5.2 User Defined Variable (UDV)

Defining a User defined Variable

To define a UDV use following syntax

Variable name=value

'**value**' is assigned to given '**variable name**' and Value must be on right side of the = sign.

Examples:

```
number=10  # this is ok
10=number  # Error, Value must be on right side of the = sign.
vech=Bus   # defines variable called 'vech' having value Bus
```

Rules for Naming variable name (Both UDV and System Variable)

1. Variable name must begin with Alphanumeric character or underscore character, followed by one or more Alphanumeric character.
   Examples of valid shell variable are:
   ```
   HOME
   SYSTEM_VERSION
   vech
   number
   ```
2. Don't put spaces on either side of the equal sign when assigning value to variable.
   For example, in following variable declaration there will be no error
   ```
   number=10
   ```
   But there will be problem for any of the following variable declaration:
   ```
   number =10
   number= 10
   number = 10
   ```
3. Variables are case-sensitive, just like filename in Linux.
   For example,
   ```
   number=10
   Number=11
   NUMBER=20
   ```
   Above all are different variable name, so to print value 20 we have to use $ echo $NUMBER and not any of the following
   ```
   echo $number     # will print 10 but not 20
   echo $Number     # will print 11 but not 20
   ```
4. You can define NULL variable as follows [NULL variable is variable which has no value at the time of definition]
   For example
   ```
   vech=
   vech=""
   ```
   Try to print it's value by issuing following command using **echo $vech**
   Nothing will be shown because variable has no value i.e. NULL variable.
5. Do not use **?,*** etc, to name your variable names.

Try the following

1. Define variable x with value 10 and print it on screen.
2. Define variable xn with value CP1 and print it on screen
3. Print sum of two numbers, let's say 6 and 3?
4. Define two variable x=20, y=5 and then to print division of x and y (i.e. x/y)

6. Decision making constructs in Shell

What computer know is 0 (zero) and 1 that is Yes or No.
To make this idea clear, lets take the help of bc - Linux calculator program. Type
bc
After this command bc is started and waiting for your commands, i.e. give it some calculation as follows type 5 + 2 as:
5 + 2
7
7 is response of bc i.e. addition of 5 + 2 you can even try
Now see what happened if you type 5 > 2
5 > 2
1
1 is response of bc, How? bc compare 5 with 2 as, Is 5 is greater than 2, bc gives the answer as 'YES' by showing 1 value. Now try
5 < 2
0
0 indicates the false i.e. Is 5 is less than 2?, the answer NO indicated by bc by showing 0.
Remember in bc logical operations always returns **true** (1) or **false** (0).
Try following in bc to clear your Idea and not down bc's response
5 > 12
5 == 10
5 != 2
5 == 5
1= < 2

Expression	Meaning to us	Your Answer	BC's Response
5 > 12	Is 5 greater than 12	NO	0
5 == 10	Is 5 is equal to 10	NO	0
5 != 2	Is 5 is NOT equal to 2	YES	1
5 == 5	Is 5 is equal to 5	YES	1
1 < 2	Is 1 is less than 2	YES	1

It means whenever there is any type of comparison in Linux Shell, it gives only one of the two answers YES or NO.

In Linux Shell Value	Meaning	Example
Zero Value (0)	Yes/True	0
NON-ZERO Value	No/False	-1, 32, 55 anything but not zero

Remember both bc and Linux Shell uses *different ways to show True/False values*

Value	Shown in bc as	Shown in Linux Shell as
True/Yes	1	0
False/No	0	Non - zero value

6.1. if condition

if condition is used for decision making in shell script. If the given condition is true then command1 is executed.

Syntax:

```
if condition
then
        command1
        ...
        ...
fi
```

Condition is defined as:

"*Condition is nothing but comparison between two values.*"

For comparison you can use test or [expression] statements.

Expression is defined as:

"An expression is nothing but combination of values, relational operator (such as >,<, <> etc) and mathematical operators (such as +, -, / etc)."

Following are all examples of expression:

5 > 2
3 + 6
3 * 65
a < b
c > 5
c > 5 + 30 -1

test command or [expression]

test command or [expression] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.

Syntax:

test expression OR [expression]

Example:

Following script determine whether given argument number is positive.

```
if test $1 -gt 0  # if [ $1 -gt 0 ] will also work
then
echo "$1 number is positive"
fi
```

Run it as follows

chmod +x ispostive
sh ispostive 5
5 number is positive
sh ispostive -45
Nothing is printed
sh ispostive
Test and justify the output

Detailed explanation

The line if test $1 -gt 0, test to see if first command line argument ($1) is greater than 0. If it is true(0) then test will return 0 and output will printed as 5 number is positive but for -45 argument there is no output because our condition is not true (no -45 is not greater than 0) hence echo statement is skipped.

For last statement we have not supplied any argument hence error **ispostive: line 1: test: -gt: unary operator expected**, is generated by shell , to avoid such error we can test whether command line argument is supplied or not.

test or [expression] works with
1.Integer (Number without decimal point)
2.File types
3.Character strings

For Mathematics, use following operator in Shell Script

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expression] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [5 -ge 6]

NOTE: == is equal, != is not equal.

For string Comparisons use

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

Shell also test for file and directory types

Test	Meaning
-s file	Non empty file
-f file	Is File exist or normal file and not a directory
-d dir	Is Directory exist and not a file
-w file	Is writeable file
-r file	Is read-only file
-x file	Is file is executable

Logical Operators
Logical operators are used to combine two or more condition at a time

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND
expression1 -o expression2	Logical OR

6.2 if...else...fi

If given condition is true then command1 is executed otherwise command2 is executed.
Syntax:

```
           if condition
             then
                           condition is zero (true - 0)
                           execute all commands up to else statement

             else
                           if condition is not true then
                           execute all commands up to fi
             fi
```

For e.g. Write Script in file isPos as follows:

```
if [ $# -eq 0 ]   # $# is used to see how many parameters passed
then
   echo "You must supply one integer"
   exit 1            # exit from the shell script
fi

if test $1 -gt 0
then
   echo "$1 number is positive"
else
   echo "$1 number is negative"
fi
```

Do the following:

```
chmod +x isPos
sh isPos 5
5 number is positive
sh isPos -45
-45 number is negative
sh isPos
 You must supply one integers
sh isPos 0
0 number is negative
```

Detailed explanation
First script checks whether command line argument is given or not, if not given then it print error message as "*You must supply one integers*". if statement checks whether number of argument

($#) passed to script is not equal (-eq) to 0, if we passed any argument to script then this if statement is false and if no command line argument is given then this if statement is true. And finally statement exit 1 causes normal program termination with exit status 1.

The last sample run **sh isPos 0** , gives output as "*0 number is negative*", because given argument is not > 0, hence condition is false and it's taken as negative number. To avoid this replace second if statement with **if test $1 -ge 0**.

7. Exercises

Exercise 1
Write a shell script to display all the files in the present working directory whose file name starts with the word given by positional parameter 1 and ends with the word given by positional parameter 2.

Exercise 2
Write a shell script to display the total number of files in your present working directory.

Exercise 3
Write a shell script to ask user to enter a command and execute it.

Exercise 4
Write a Shell script which accepts two numbers a and b from the user directly and displays the result of a^2+b^2. The shell script also takes a parameter. This parameter is added to previously computer (**a^2+b^2**) value and displays the final result.

Exercise 5
Write a Shell script which accepts three numbers a, b and c from the user directly and displays the result of ax^2+bx+c. The shell script takes the value of x as a parameter.

Exercise 6
Write a program to find the largest of 2 Numbers

(A) Take Numbers from user using read command
(B) Take Numbers from user as command line arguments

Exercise 7
Write a program to find the largest of 3 Numbers

(A) Take Numbers from user using read command
(B) Take Numbers from user as command line arguments

Exercise 8
Take radius as user input and calculate Area and Circumference of a circle if radius is greater than 5

ADDITIONAL QUESTIONS

Question #1

Write a shell script program named Lab3_1.sh which accepts a string from the user. The shell script program should print the message "The entered string <string> is a directory under the current working directory" if the string user entered is a directory under the present working directory. Else the program should print the message "The entered string <string> is NOT a directory under the present working directory".

Question #2

Write a shell script program named Lab3_2.sh to take 3 sides of a triangle as command line argument. Find whether the inputs can form a triangle. If yes, then find the perimeter of the triangle and check whether the triangle is a scalene triangle. Display appropriate messages for invalid data [like negative numbers, not a triangle etc ..].

Note: If A, B and C are three sides of a triangle, then the necessary condition for a valid triangle is

A + B > C and A – B < C

Exercise Solutions

```
ex 1: ls -dp "$1"*"$2"| grep -v /

ex 2 : ls -l | wc -l

ex 3 : echo "Enter a command"
            read cmd
            $cmd

ex 4: echo 'enter value of a :'
         read a
         echo 'enter value of b :'
         read b
         g=`expr $a \* $a`
         h=`expr $b \* $b`
         sum=`expr $g + $h`
         echo $sum
         x=$1
         echo finalsum=`expr $sum + $x`
```

```
ex 5:echo 'enter the value of a:'
     read a
     echo 'enter the value of b:'
     read b
     echo 'enter value of c :'
     read c
     x=$1
     g=`expr $a \* $x \* $x`
     h=`expr $b \* $x`
     echo result is :`expr $g + $h + $c`

ex 6: (A)
     echo 'enter first number :'
     read a
     echo 'enter second number:'
     read b
     if test $a -gt $b
     then
          echo $a is greater than $b
     else
          echo $b is greater than $a
     fi
(B)
     a=$1
     b=$2
     if test $a -gt $b
     then
          echo $a is greater than $b
     else
          echo $b is greater than $a
     fi

ex 7: (A)
     echo 'enter first number:'
     read a
     echo 'enter second number:'
     read b
     echo 'enter third number:'
     read c
     if test $a -gt $b -a $a -gt $c
     then
          echo $a is the largest
     elif [ $b -gt $c ]
     then
          echo $b is the largest
     else
          echo "$c is the largest"
     fi
```

(B)

```
     a=$1
     b=$2
     c=$3
     if test $a -gt $b -a $a -gt $c
     then
          echo $a is the largest
     elif [ $b -gt $c ]
     then
          echo $b is the largest
     else
          echo $c is the largest
     fi
```

ex 8:

```
     echo 'enter radius :'
     read radius
     if test $radius -gt 5
     then
          echo area is : `expr $radius \* $radius \* 3`
          echo circumference is : `expr 2 \* 3 \* $radius`
     else
          echo 'enter radius greater than 5'
     fi
```

ADDITIONAL QUESTIONS SOLUTIONS

A1.

```
     echo "Enter the string"
     read str
     n=`expr $(ls */| grep / | grep -c $str)`
     if [ $n -eq 0 ]
     then
          echo "The entered string $str is not a directory under
     the current working directory"
     else
          echo "The entered string $str is a directory under the
     current working directory"
     fi
```

A2:

```
     if [ $# -eq 0 ]
     then
          echo "You must supply a value"
     else
          if [ $1 -lt 0 -o $2 -lt 0 -o $3 -lt 0 ]
          then
               echo "Supply Positive Numbers"
          elif [ `expr $1 + $2` -gt $3 -a `expr $1 - $2` -lt $3
     ]
          then
               echo "Valid Triangle"
               echo "Perimeter = `expr $1 + $2 + $3`"
               if [ $1 -eq $2 -a $1 -eq $3 ]
               then
                    echo "Equilateral Triangle"
               elif [ $1 -eq $2 -o $1 -eq $3 -o $2 -eq $3 ]
               then
                    echo "Isosceles Triangle"
               else
                    echo "Scalene Triangle"
               fi
          else
               echo 'invalid values'
          fi
     fi
```