Birla Institute of Technology & Science, Pilani, K. K Birla Goa Campus
Computer Programming (CS F111)
Second Semester 2013-2014
Lab-04 (UNIX- Shell scripting)

Objectives

1. Introduction
2. More Decision Making Constructs in Shell
3. Looping Constructs in Shell
4. Introduction to Arrays
5. Math Library Functions
6. Exercises
7. Additional Exercises
8. Exercise Answers

---

1. Introduction

In the last lab (lab 3) we learned how to write an interactive shell script, how to use system defined variables and how to define and use user defined variables. We also learned the usage one of the decision making constructs. In this lab we are going to focus on more powerful decision making constructs available in Linux shell and how to use them to write a powerful shell script.

2. More Decision making constructs in Shell

2.1. Nested if-else-fi

You can write the entire if-else construct within either the body of if statement of the body of else statement. This is called the nesting of ifs. Consider a file named **isPos_nestedif** with the following content.

```
if [ $# -eq 0 ]
then
    echo "You must supply one integer"
else
        if test $1 -gt 0
        then
           echo "$1 number is positive"
        else
           echo "$1 number is negative"
        fi
fi
```

Run the above shell script as follows:

```
chmod +x isPos_nestedif
sh isPosnestedif 20
20 number is positive
```

Note that Second *if-else* construct is nested in the first *else* statement. If condition in the first '*if*' statement is false the condition in the second '*if*' statement is checked. If it is false as well the final *else* statement is executed.
You can use the nested *if*s as follows also:
Syntax:

```
if condition
then
        if condition
        then
                do this
        else
                do this
        fi
else

                do this
fi
```

2.2. Multilevel if-then-else

Syntax:

```
    if condition
        then
                condition is zero (true - 0)
                execute all commands up to elif statement
        elif condition1
        then
                condition1 is zero (true - 0)
                execute all commands up to elif statement
        else
                None of the above condition, condtion1, condtion2 are true (i.e.
                all of the above nonzero or false) execute all commands up to fi
        fi
```

For multilevel if-then-else statement try the following script in a file named **isPos_Multilevelif**:

```
if [ $# -eq 0 ]
then
    echo "You must supply one integer"
elif test $1 -gt 0
then
   echo "$1 number is positive"
elif test $1 –lt 0
then
   echo "$1 number is negative"
elif test $1 –eq 0
then
   echo "$1 number is zero"
else
   echo “Opps! $1 is Not a Number, supply a Number”
fi
```

Try above script as follows:

```
chmod +x isPos_Multilevelif
sh isPos_Multilevelif 20
sh isPos_Multilevelif 0
sh isPos_Multilevelif -20
sh isPos_Multilevelif CP
```

2.3. The case Statement

The case statement is good alternative to multilevel if-then-else-fi statement. It enables you to match several values against one variable. It's easier to read and write.

Syntax:

```
case  $variable-name  in
    pattern1)    command
                 command;;
    pattern2)    command
                 command;;
    patternN)    command
                 command;;
          *)     command
                 command;;
  esac
```

The *$variable-name* is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and it's executed if no match is found. For example write script named **vehicle** as follows:

```
rental=$1
case $rental in
"car") echo "For $rental Rs.400 per day";;
"van") echo "For $rental Rs.500 per day";;
"jeep") echo "For $rental Rs.300 per day";;
"bicycle") echo "For $rental Rs.20 per day";;
*) echo "Sorry, I cannot get a $rental for you";;
esac
```

Save it and run it as follows:

```
chmod +x vehicle
sh vehicle van
sh vehicle car
sh vehicle Maruti-800
```

Note that esac is always required to indicate end of case statement.

3. Looping Constructs in Shell

Loop defined as:

"A Program *can repeat particular set of instructions again and again, until particular condition satisfies. A group of instructions that is executed repeatedly is called a loop.*"

Bash supports:

- for loop
- while loop

Note that in each and every loop,

(a) First, the variable used in loop condition must be initialized, then execution of the loop begins.
(b) A test (condition) is made at the beginning of each iteration.
(c) The body of loop ends with a statement that modifies the value of the test (condition) variable.

3.1. for Loop

Syntax :

```
for { variable name } in { list }
do
        execute one for each item in the list until the list is not finished
        (And repeat all statement between do and done)
done
```

For example write script named **display_num** as follows:

```
for i in 1 2 3 4 5
do
   echo "Welcome $i times"
done
```

Run the above script as follows:

```
chmod +x display_num
sh display_num
```

The for loop first creates i variable and assigned a number to i from the list of number from 1 to 5, The shell execute echo statement for each assignment of i. (This is usually know as iteration) This process will continue until all the items in the list were not finished, because of this it will repeat 5 echo statements.

You can use following syntax as well:

Syntax:

```
for (( expr1; expr2; expr3 ))
do
    repeat all statements between do and
    done until expr2 is TRUE
done
```

In above syntax BEFORE the first iteration, ***expr1*** is evaluated. This is usually used to initialize variables for the loop.
All the statements between do and done is executed repeatedly UNTIL the value of ***expr2*** is TRUE.
After each iteration of the loop, ***expr3*** is evaluated. This is usually used to increment a loop counter.

For example write script named **display_num1** as follows:

```
for (( i=0 ; i<=5; i++ ))
do
  echo "Welcome $i times"
done
```

Run the above script as follows:

```
chmod +x display_num1
bash display_num1
```

In above example, first expression (i=0), is used to set the value variable **i** to zero. Second expression is condition i.e. all statements between do and done executed as long as expression 2 (i.e. continue as long as the value of variable **i** is less than or equal to 5) is TRUE. Last expression **i++** increments the value of **i** by 1 i.e. it's equivalent to i=i+1 statement.

3.2 Nesting of for Loop

Loop statement can also be nested similar to the if statements. You can nest the for loop. To understand the nesting of for loop see the following shell script named **display_num_nested**.

```
for (( i=1; i<=5; i++ ))        # Outer for loop
  do
    for (( j=1; j<=5; j++ ))  # Inner for loop
    do
        echo  "$i,$j   "
    done
   echo ""                    #print the new line
  done
```

Run the above script as follows:

```
chmod +x display_num_nested
bash display_num_nested
```

Here, for each value of **i** the inner loop is cycled through 5 times, with the variable **j** taking values from 1 to 5. The inner for loop terminates when the value of **j** exceeds 5, and the outer loop terminates when the value of **i** exceeds 5.

Infinite loop

Infinite for loop can be created with empty expressions, such as

```
for (( ; ; ))
do
  echo "infinite loops [ hit CTRL+C to stop]"
done
```

Conditional exit with break

You can do early exit with break statement inside the for loop. You can exit from within a FOR or WHILE loop using break. General break statement inside the for loop:

```
for I in 1 2 3 4 5
do
        statements1     #Executed for all values of "I", up to distater condition
 if (disaster-condition)
 then
         break          #Abandon the loop.
 fi
         statements3    #While good and, no disaster-condition.
done
```

Early continuation with continue statement

To resume the next iteration of the enclosing FOR, WHILE or UNTILloop use continue statement.

```
for I in 1 2 3 4 5
do
        statements1  #Executed for all values of "I", up to a disaster-condition
        statements2
 if (condition)
 then
         continue   #Go to next iteration of I in the loop and skip statements3
 fi
        statements3
done
```

3.3. while loop

Syntax:

```
 while [ condition ]
 do
    command1
    command2
    command3
done
```

The loop is executed as long as given condition is true.
For example write script named **mul_Table** as follows:

```
if [ $# -eq 0 ]
then
    echo "Error - Number missing form command line    argument"
exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
    echo "$n * $i = `expr $i \* $n`"
    i=`expr $i + 1`
done
```

Run the above script as follows:
chmod +x mul_Table

sh mul_Table 7

Above loop can be explained as follows:

n=$1	Set the value of command line argument to variable n. (Here it's set to 7)
i=1	Set variable i to 1
while [$i -le 10]	This is our loop condition, here if value of i is less than 10 then, shell execute all statements between do and done
do	Start loop
echo "$n * $i = `expr $i \* $n`"	Print multiplication table as 7 * 1 = 7 7 * 2 = 14 .... 7 * 10 = 70, Here each time value of variable n is multiply by i.
i=`expr $i + 1`	Increment i by 1 and store result to i. (i.e. i=i+1) **Caution:** If you ignore (remove) this statement than our loop become infinite loop because value of variable i always remain less than 10 and program will only output 7 * 1 = 7 ... ... E (infinite times)
done	Loop stops here if i is not less than 10 i.e. condition of loop is not true. Hence loop is terminated

Infinite loops

Infinite for while can be created with empty expressions, such as

```
while :
do
        echo "infinite loops [ hit CTRL+C to stop]"
done
```

Conditional while loop exit with break statement

You can do early exit with the break statement inside the whil loop. You can exit from within a WHILE using break.

In this example, the break statement will skip the while loop when user enters -1, otherwise it will keep adding two numbers:

```
while :
do
        read -p "Enter two numnbers ( - 1 to quit ) : " a b
        if [ $a -eq -1 ]
        then
                break
        fi
        ans=$(( a + b ))
        echo $ans
done
```

Early continuation with the continue statement

To resume the next iteration of the enclosing WHILE loop use the continue statement as follows:

```
while [ condition ]
do
 statements1   #Executed as long as condition is true and/or, up to a disaster-
               condition if any
 statements2
 if (condition)
 then
     continue   #Go to next iteration of I in the loop and skip statements3
 fi
 statements3
done
```

4 Introduction to Arrays

An array is a variable containing multiple values may be of same type . There is no maximum limit to the size of an array, nor any requirement that member variables be indexed or assigned contiguously. Array index starts with zero. To run all scripts related to arrays, use bash filename.

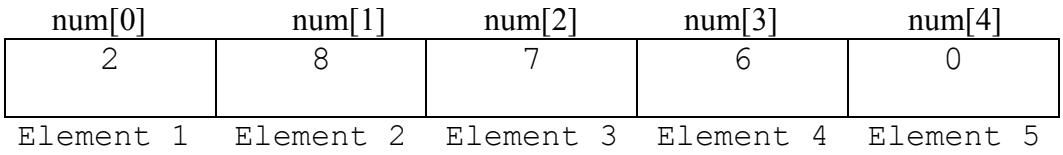

Figure A Example of array

4.1 Declaring an Array and Assigning values

In bash, array is created automatically when a variable is used in the format like,

name[index]=value

- name is any name for an array
- Index could be any number or expression that must evaluate to a number greater than or equal to zero. You can declare an explicit array using declare -a arrayname.

```
arraymanip
Unix[0]='Debian'
Unix[1]='Red hat'
Unix[2]='Ubuntu'
Unix[3]='Suse'
or
Unix=('Debian' 'Red hat' 'Ubuntu' 'Suse' 'Fedora' 'UTS' 'OpenLinux');
echo ${Unix[1]}
output:
Red hat
```

To access an element from an array use curly brackets like ${name[index]}.

4.2 Print the Whole Bash Array

There are different ways to print the whole elements of the array. If the index number is @ or *, all members of an array are referenced. You can traverse through the array elements and print it, using looping statements in bash.

```
echo ${Unix[@]}
output:
Debian Red hat Ubuntu Suse
```

Referring to the content of a member variable of an array without providing an index number is the same as referring to the content of the first element, the one referenced with index number 0.

4.3 Length of the Bash Array

We can get the length of an array using the special parameter called $#.

${#arrayname[@]} gives you the length of the array.

4.4 Length of the nth Element in an Array

${#arrayname[n]} should give the length of the nth element in an array

```
echo ${#Unix[3]} # length of the element located at index 3 i.e Suse
output:
bash arraymanip
4
```

4.5 Extraction by offset and length for an array

The following example shows the way to extract 2 elements starting from the position 3 from an array called Unix.

```
echo ${Unix[@]:3:2}
output:
Suse Fedora
```

The above example returns the elements in the 3rd index and fourth index. Index always starts with zero.

4.6 Extraction with offset and length, for a particular element of an array

To extract only first four elements from an array element . For example, Ubuntu which is located at the second index of an array, you can use offset and length for a particular element of an array.

```
echo ${Unix[2]:0:4}
output:
Ubun
```

The above example extracts the first four characters from the 2nd indexed element of an array.

4.7 Search and Replace in an array elements

The following example, searches for Ubuntu in an array elements, and replace the same with the word 'SCO Unix'.

```
echo ${Unix[@]/Ubuntu/SCO Unix}
output:
Debian Red hat SCO Unix Suse Fedora UTS OpenLinux
```

In this example, it replaces the element in the 2nd index 'Ubuntu' with 'SCO Unix'. But this example will not permanently replace the array content.

4.8 Add an element to an existing Bash Array

The following example shows the way to add an element to the existing array.

```
Unix=("${Unix[@]}" "AIX" "HP-UX")
```

In the array called Unix, the elements 'AIX' and 'HP-UX' are added in 7th and 8th index respectively.

4.9 Remove an Element from an Array

unset is used to remove an element from an array.unset will have the same effect as assigning null to an element.

```
unset Unix[3]
echo ${Unix[3]}
```

The above script will just print null which is the value available in the 3rd index. The following example shows one of the way to remove an element completely from an array.

4.10 Remove Bash Array Elements using Patterns

In the search condition you can give the patterns, and stores the remaining element to an another array as shown below.

```
Unix=('Debian' 'Red hat' 'Ubuntu' 'Suse' 'Fedora');
patter=( ${Unix[@]/Red*/} )
echo ${patter[@]}
output:
Debian Ubuntu Suse Fedora
```

The above example removes the elements which has the patter Red*.

4.11 Copying an Array

Expand the array elements and store that into a new array as shown below.

```
Unix=('Debian' 'Red hat' 'Ubuntu' 'Suse' 'Fedora' 'UTS' 'OpenLinux');
Linux=("${Unix[@]}")
echo ${Linux[@]}
output:
Debian Red hat Ubuntu Fedora UTS OpenLinux
```

4.12 Concatenation of two Bash Arrays

Expand the elements of the two arrays and assign it to the new array.

```
Unix=('Debian' 'Red hat' 'Ubuntu' 'Suse' 'Fedora' 'UTS' 'OpenLinux');
Shell=('bash' 'csh' 'jsh' 'rsh' 'ksh' 'rc' 'tcsh');
UnixShell=("${Unix[@]}" "${Shell[@]}")
echo ${UnixShell[@]}
echo ${#UnixShell[@]}
output:
Debian Red hat Ubuntu Suse Fedora UTS OpenLinux bash csh jsh rsh ksh rc tcsh
```

It prints the array which has the elements of the both the array 'Unix' and 'Shell', and number of elements of the new array is 14.

4.13 Deleting an Entire Array

unset is used to delete an entire array.

```
unset Unix
echo ${#UnixShell[@]}
output:
0
```

After unset an array, its length would be zero as shown above.

5. Math library functions

If `bc` is invoked with the `-l` option, a math library is preloaded and the default scale is set to 20. The math functions will calculate their results to the scale set at the time of their call. The math library defines the following functions:

`s (x)`
: The sine of *x*, *x* is in radians.

`c (x)`
: The cosine of *x*, *x* is in radians.

`a (x)`
: The arctangent of *x*, arctangent returns radians.

`l (x)`
: The natural logarithm of *x*.

e (*x*)

The exponential function of raising *e* to the value *x*.

Example
The following will assign the value of "pi" to the shell variable *pi*.
pi=$(echo "scale=10; 4*a(1)" | bc -l)
scale=10 means value of pi will have 10 digit precision.

Output:
3.1415926532

6. Exercises

Try out the following Programs

1. **Write a shell script program to find sum of first N numbers (take N from the user).**
2. **Write a shell script program to print a given number in reverse order, for example, if the number is 12345 it must print as 54321.**
3. **Sort the given integer numbers in ascending order using array.**
4. **Calculate the average of given integer numbers on the command line.**
5. **Write a shell script program to find factorial of a positive integer N (take N from the user). Remember that 0!=1, fact(N)=N×fact(N-1)! or fact(N)=1×2×…..×N.**
6. **Write a shell script program to print given numbers sum of all digits, for example, if number is 12345 the sum of all digits will be 1+2+3+4+5 = 15.**
7. **Write a script to display all the prime numbers between 3 and N, where N is given as command line argument.**

7. Additional Exercises

1. Write a script to find the given number is an Armstrong number of 3 digits. It is Also known as narcissistic numbers, Armstrong numbers are the sum of their own digits to the power of the number of digits. As that is a slightly brief wording, let me give an example: $153 = 1^3 + 5^3 + 3^3$
Each digit is raised to the power three because 153 has three digits. They are totalled and we get the original number again!

2. Write a script to print the Fibonacci series (Given, f0=f1=1, and fn=fn-1+fn-2, n≥2) between 1 and N where N is taken (from the user) by using read. If N =10 then the output of your program should be : 1, 1, 2, 3, 5, 8, 13, 21, 34.

3. Write a script to display the following pattern up to N lines(take N from the user). N is 10 in below example

8.Exercise Answers

1. Script to find the sum of first N numbers.

```
echo "Enter a number: "
read num
i=1
sum=0

while [ $i -le $num ]
do
sum=`expr $sum + $i`
i=`expr $i + 1`
done
echo "The sum of first $num numbers is: $sum"
```

2. Script to reverse a given positive integer

```
if [ $# -ne 1 ]
then
   echo "Usage: $0   number"
   echo "       I will find reverse of given positive integer"
   echo "       For eg. $0 123, I will print 321"
   exit 1
fi

n=$1
rev=0; sd=0

while [ $n -gt 0 ]
do
   sd=`expr $n % 10`
```

```
    rev=`expr $rev \* 10  + $sd`
    n=`expr $n / 10`
done
echo  "Reverse number is $rev"
```

3. Sort the given five integer numbers in ascending order (using array)

```
# Declare the array of 5 subscripts to hold 5 numbers
nos=(4 -1 2 66 10)

# Prints the number before sorting
echo "Original Numbers in array:"
for (( i = 0; i <= 4; i++ ))
do
  echo ${nos[$i]}
done

# Now do the Sorting of numbers
for (( i = 0; i <= 4 ; i++ ))
do
   for (( j = $i; j <= 4; j++ ))
   do
      if [ ${nos[$i]} -gt ${nos[$j]}  ]; then
          t=${nos[$i]}
          nos[$i]=${nos[$j]}
          nos[$j]=$t
      fi
   done
done

# Print the sorted number
echo -e "\nSorted Numbers in Ascending Order:"
for (( i=0; i <= 4; i++ ))
do
  echo ${nos[$i]}
done
```

4. Calculating average of given integer numbers on command line arguments

```
avg=0
temp_total=0
number_of_args=$#

# First see the sufficent cmd args
if [ $# -lt 2 ] ; then
   echo -e "Oops! I need atleast 2 command line args\n"
   echo -e "Syntax: $0: number1 number2 ... numberN\n"
```

```
  echo -e "Example:$0 5 4\n\t$0 56 66 34"
  exit 1
fi

# now calculate the average of numbers given on command line as cmd args
for i in $*
do
  #  addition of all the numbers on cmd args
  temp_total=`expr $temp_total + $i `
done

avg=`expr $temp_total / $number_of_args `
echo "Average of all number is $avg"
```

5. Calculating factorial of a given integer number

```
n=0; on=0
fact=1
echo -n "Enter number to find factorial : "
read n

on=$n
while [ $n -ge  1 ]
do
 fact=`expr $fact \* $n`
 n=`expr $n - 1`
done
echo "Factorial for $on is $fact"
```

6 Sum of all the digits of given number.

```
echo enter any number
read n

sum=0
sd=0
while [ $n -gt 0 ]
 do
 sd=`expr $n % 10`
 sum=`expr $sum + $sd`
 n=`expr $n / 10`
done

 echo "sum of digit for number is $sum”
```

7 All prime numbers between 3 and given number

```
#!/bin/bash
```

```
echo “Enter the number”
read j
echo "1"
i=3
flag=0
tem=2

while [ $i -ne $j ]
do
	temp=`echo $i`

	while [ $temp -ne $tem ]
	do
		temp=`expr $temp - 1`
		n=`expr $i % $temp`

		if [ $n -eq 0 -a $flag -eq 0 ]
		then
			flag=1
		fi
	done

	if [ $flag -eq 0 ]
	then
		echo $i
	else
		flag=0
	fi
	i=`expr $i + 1`
done
```