LAB NO: 3 **Date:**

UNIX SHELL PROGRAMMING (SHELL SCRIPTING)

Objectives:

1. To recall the Unix shell programming.
2. To identify System variables.

1. The Unix shell programming

Shell programming is a group of commands grouped together under single filename. After logging onto the system a prompt for input appears which is generated by a Command String Interpreter program called the shell. The shell interprets the input, takes appropriate action, and finally prompts for more input. The shell can be used either interactively - enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled.

2. shbang line, comments, wildcards and keywords

The shbang line The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #! followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.

E*XAMPLE* #!/bin/sh

Comments Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.

E*XAMPLE* # this text is not interpreted by the shell

Wildcards There are some characters that are evaluated by the shell in a special way. They are called shell meta characters or "wildcards". These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

E*XAMPLE*

Filename expansion:

```
rm *; ls ??; cat file[1-3];
```

Quotes protect metacharacters:

```
echo "How are you?"
```

Shell keywords :

Some of the shell keywords are echo, read, if fi, else, case, esac, for, while, do, done, until, set, unset, readonly, shift, export, break, continue, exit, return, trap , wait, eval ,exec, ulimit , umask.

3. shell variables, expressions and statements

Shell variables change during the execution of the program.

Variable naming rules:

- A variable name is any combination of alphabets, digits and an underscore (“-“);
- No commas or blanks are allowed within a variable name.
- The first character of a variable name must either be an alphabet or an underscore.
- Variables names should be of any reasonable length.
- Variables name are case sensitive. That is, Name, NAME, name, Name, are all different variables.

Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.

EXAMPLE

```
variable_name=value
name="John Doe"
x=5
```

Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.

EXAMPLE

```
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:.
export PATH
```

Extracting values from variables To extract the value from variables, a dollar sign is used.

EXAMPLE [here, echo command is used display the variable value]

```
echo $variable_name
echo $name
echo $PATH
```

4. Shell input and output

Input:

To get the input from the user *read* is used.

Syntax : *read* x y #no need of commas between variables

The read command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The read command can accept multiple variable names. Each variable will be assigned a word. No need to declare the variables to be read from user

Output :

echo can be used to display the results. Wildcards must be escaped with either a backslash or matching quotes.

Syntax :

echo "Enter the value of b" (or) echo Value of b is $b(for variable).

5. Basic Arithmetic operations

The shell does not support arithmetic operations directly (ex: a=b+c). UNIX/Linux commands must be used to perform calculations.

Command	**Syntax**	**Example**
expr	*expr* expression operators: + , - , /, %,=,==,!=	*a=$(expr $a + 1)* *a=`expr $a + 1`* space should not be present between = and *expr*. Space should be present between operator and operands. To access values $ has to be used for operands. Performs only integer arithmetic operations.

test [ ]	[condition/expression] Note one space should be present after [and before]. Also operand and operator must be separated by a space. *operators*: **Integers**: -eq, -ne, -gt,-lt,-ge, -le, ==, != **Boolean**: !, -o(or), -a(and) **String**: =, !=,-z(zero length), -n(non-zero length), [$str] (true if $str is not empty) **File**: -f (ordinary file), -d (directory), -r (readable), -w, -x, -s (size is zero), -e (exists)	*echo "Enter Two Values"* *read a b* *result=$[a == b]* *echo "Check for Equality $result"* **O/P:** Enter Two Values 4 4 Check for Equality 1 *test* works in combination with control structures refer *section 6*.
test (( ))	Performs integer arithmetic. Here spacing does not matter also we need not include $ for the variables. Useful in performing increment or detriment operations.	*echo "Enter the two values"* *read a b* *echo "enter operator(+, -, /, % *)"* *read op* *((a++))* *result=$((a $op b))* *echo "Result of performing $a $op $b is $result"* **O/P:**Enter the two values 4 6 enter operator(+, -, /, % *) * Result of performing 5 * 6 is 30

bc	refer section 4 of Lab 2. *bc* can be used to perform floating point operations.	*echo "Enter the two values"* *read a b* *echo "Enter operator(+, -, /, % *)"* *read op* *result=`bc -l <<<$a\$op$b`* # or use result=`*echo "$a $op $b" | bc -l*` *echo " Result of performing $a $op $b is $result"* **O/P:**Enter the two values 4 5 Enter operator(+, -, /, % *) * Result of performing 4 * 5 is 20

6. Control statements

The shell control structure is similar to C syntax, but instead of brackets {} statements like *then-fi* or *do-done* are used. The *then*, *do* has to be used in next line, otherwise *;* has to be used to mark the next line.

Control Structure	**Syntax**	**Example**
if	*if* condition ; *then* command(s) *fi* *OR* *if* condition *then* command(s) *fi*	*read character* *if ["$character" = "2"]; then* *echo " You entered two."* *fi* **O/P:** 2 You entered two

if else	*if* condition ; *then* command(s) *else* command(s) *fi*	*read fileName* *if [-e $fileName]; then* *echo " File $fileName exists"* *else* *echo " File $fileName does not exist"* *fi* **O/P:** LAB3.sh File LAB3.sh exists
else if ladder	*if* condition ; *then* command(s) *elif* condition ; *then* command(s) *fi*	*read a b* *if [$a == $b]; then* *echo "$a is equal to $b"* *elif [$a -gt $b]; then* *echo "$a is greater than $b"* *elif ((a<b)) ; then* *echo "$a is less than $b"* *else* *echo "None of the condition met"* fi **O/P:** 4 5 4 is less than 5
switch case	*case* word *in* pattern1) command(s) ;; pattern2) command(s) ;; ... *) command(s) ;; *esac*	*echo -n "Enter a number 1 or string Hello or character A"* *read character* *case $character in* *1) echo "You entered one.";;* *"Hello") echo -n "You entered two."* *echo "Just to show multiple commands";;* *'A') echo "You entered three.";;* **) echo "You did not enter a number"* *echo "between 1 and 3."* *esac* **O/P:** Enter a number 1 or string Hello or character A: Hello You entered two. Just to show multiple commands

for	*for* ((initialization; condition; expo)); *do* command(s) *done*	*read n* *for ((i=1; i<=n; i++));do* *echo -n $i* *done* **O/P:** 5 12345
for each	*for* variable in list *do* command(s) *done*	*IFS=$'\n'* #field separator is \n instead of default space *x=`ls -l | cut -c 1`* *for i in $x;do* *if [$i = "d"] ; then* *echo "This is the directory"* *fi* *done* **O/P:** $ls -l -rw-r--r-- 1 … script.sh -rw-r--r-- 1 … file2.txt drwxr-xr-x 2 … test $bash script.sh This is the directory
while	while condition do command(s) to be executed while the condition is true done	read n i=1; while ((i <= n)); do echo -n $i " " ((i++)) done echo “" O/P: 5 1 2 3 4 5
until	until condition do command(s) to be executed until condition is true i.e while the condition is false. done	read n i=1 until ((i > n)); do echo -n $i " " ((i++)) done O/P: 5 1 2 3 4 5

exit	exit num command may be used to deliver an num exit status to the shell (num must be an integer in the 0 - 255 range).	echohi echo "last error status $?" exit $? #exit the script with las error status echo "HI" # never printed O/P: echohi: command not found last error status 127

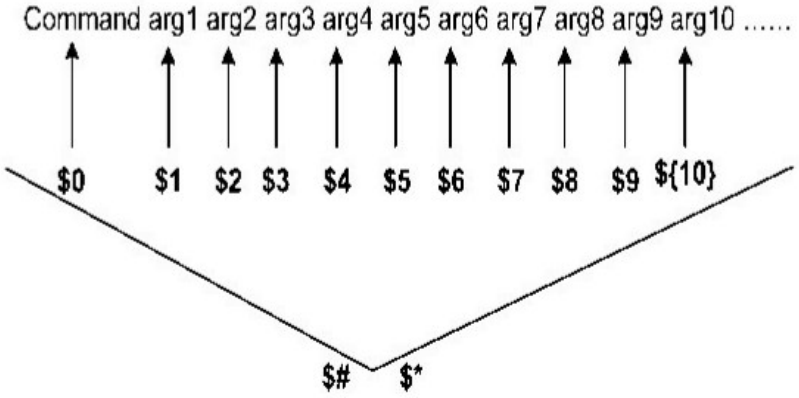
7. Execution of a shell script

Prepare the shell script using either *text editor* or *vi*. After preparing the script file in use *sh* or *bash* command to execute a shell script. Example: $*bash test.sh* [Here the test.sh is the file to be executed]. OR give executable permission to the script and run ./scriptName. Example: *$chmod +x test.sh*

$./test.sh

3. Command Line arguments:

Command line arguments (also known as positional parameters) are the arguments specified at the command prompt with a command or script to be executed. The locations at the command prompt of the arguments as well as the location of the command, or the script itself, are stored in corresponding variables. These variables are special shell variables.



Positional parameter	Description
$0	The command or script name
$#	Total number of arguments.

$1 to $9	Arguments 1 through 9
${10} and so on	Arguments 10 and further
$*	All the arguments
$$	PID of the running script
$@	Returns a sequence of strings (``$1", ``$2", ... ``$n''). Same as $* when unquoted. $@ interprets each quoted argument as a separate argument. for i in "$@"; do echo $i # loop $# times done for i in "$*";do echo $i # loop 1 times done

EXAMPLE:

At the command line:

$scriptname arg1 arg2 arg3 ...

Inside script:

```
echo $1 $2 $3 ${10}   #Positional parameters
echo $*               #All the positional parameters
echo $#               #The number of positional parameters
shift
```

4. System variables

When you log in on UNIX, your current shell (login shell) sets a unique working environment for you which is maintained until you log out. You can see system variables by giving command like $ set, few of the important system variables are

System Variable	value	Meaning
BASH	/bin/bash	Our shell name
BASH_VERSION	1.14.7(1)	Our shell version name

COLUMNS	80	No. of columns for our screen
HOME	/home/vivek	Our home directory
LINES	25	No. of columns for our screen
LOGNAME	students	Our logging name
OSTYPE	Linux	Our os type
PATH	/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1	[\u@\h \W]\$	Our prompt settings
PWD	/home/students/Common	Our current working directory
SHELL	/bin/bash	Our shell name
USER	vivek	User name who is currently login to this PC

NOTE that Some of the above settings can be different in your PC. You can print any of the above variables contain as follows

$ echo $USER

$ echo $HOME

[Caution: Do not modify System variable this can some time create problems.]

5. Arrays and Functions

10.1 Arrays: An array variable that can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

Array Declaration

If you are using **ksh** shell the here is the syntax of array initialization:

set -A array_name value1 value2 ... valuen

If you are using **bash** shell the here is the syntax of array initialization:

array_name=(value1 ... valuen) or

declare -a array_name

Accessing Array values

After you have set any array variable, you access it as follows:

${array_name[index]}

Here array_name is the name of the array, and index is the index of the value to be accessed.

Example:

```
read -a inputArrayOfNumbers  # input separated by spaces and not by carriage return
echo -n "Entered input is…"
for i in ${inputArrayOfNumbers[@]} ; do
echo -n $i " "
done
```

O/P:

5 4 45 3
Entered input is…5 4 45 3

10.2 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. Using functions to perform repetitive tasks is an excellent way to create code reuse.

Function Definition

To define a function, simply use the following syntax:

```
function_name () {  # can also use function function_name
  list of command(s)
# to use parameters passed to the function use $1, $2…
}
```

Calling a Function

To call the function in the script use the following syntax:

```
function_name Arg1 arg2… # without spaces
```

Returning values from function

Exit status can be returned from the function using *return* statements in the function definition.

Example:

```
function_name () {
  list of command(s)
   retval=0
  return “$retval"
```

In the main routine the values can be retrieved using *$?*.

Example:

```
echo The previous function has a return value of $?
retval=$?  #  to get the exit status of the function.
```

Variables can be defined outside the scope of the any function, so that they can be shared among the functions and main routine. To return the string *echo* can be used in the function definition and use retval=$(*function_name*) in the main routine.

Arrays as parameter to function

An array can be passed as a parameter to the function as normal variable, while in the definition it can be accessed using *$@*.

Example:

```
myFunction() {
param1=("${!1}")
param2=("${!2}")
for i in ${param1[@]}; do
for j in ${param2[@]}; do
if [ "${i}" == "${j}" ]; then
echo ${i}
echo ${j}
fi
done
done
}
a=(foo bar baz)
b=(foo bar qux)
myFunction a[@] b[@] # would display foo foo bar bar.
```

Lab exercises

1. Write a shell script to find whether a given file is the directory or regular file.
2. Write a shell script to list all files(only file names) containing the input pattern(string) in the folder entered by the user.
3. Write a shell script to replace all files with .txt extension with .text in the current directory. This has to be done recursively i.e if the current folder contains a folder with abc.txt then it has to be changed to abc.text (Hint: use find, mv)
4. Write a shell script to calculate the gross salary. GS=Basics + TA + 10% of Basics. Floating point calculations has to be performed.
5. Write a program to copy all the files(having file extension input by the user) in the current folder to the new folder input by the user. ex: user enter .text TEXT then all files with .text should be moved to TEXT folder. This should be done only at single level. i.e if the current folder contains a folder name ABC which has .txt files then these files should not be copied to TEXT.

6. Write a shell script to modify all occurrences of "ex:" with "Example:" in all the files present in current folder if "ex:" occurs at the start of the line or after a period (.) . Example: if a file contains a line: ex: "this is first occurrence so should be replaced" and second ex: "should not be replaced as it occurs in the middle of the sentence."
7. Write a shell script to make a duplicate copy of a specified file through command line.
8. Write a shell script to remove all files that are passed as command line arguments interactively.
9. Write a program to sort the strings that are passed as a command line arguments. (ex: ./script.sh "OS Lab" "Quoted strings" "Command Line" "Sort It". The output should be "Command Line" "OS Lab" "Quoted strings" "Sort It". (make use of usrdefined sort function)
10. Implement *wordcount* script that takes *-linecount*, *-wordcount*, *-charcount* options and performs accordingly, on the input file that is passed as command line argument (use case statement)
11. Write a menu driven shell script to read list of patterns as command line arguments and perform following operations.
    a. Search the patterns in the given input file. Display all lines containing the patten in the given input file.
    b. Delete all occurances of the pattern in the given input file.
    c. Exit from the shell script.

Additional Exercises

1. Write a shell script to check whether the user entered number is prime or not.
2. Write a shell script to find the factorial of number.
3. Write a shell script to input a file and display permissions of the owner group and others.
4. Write a shell script to display all files that are created between the input years range.(ex with 2014-2015)

Write a shell script that accepts a file name starting and ending line numbers as arguments and displays all the lines between the given line numbers.