CHARUTAR VIDYA MANDAL'S
SEMCOM
Vallabh Vidyanagar

Faculty Name: Ami D. Trivedi
Class: TYBCA (Semester-V)
Subject: US05EBCA01 (Basics of UNIX Operating System)

*UNIT – 3 (Introduction to Shell Scripting)

SHELL SCRIPTS

Shell offers the facility of storing a group of commands in a file and then executing the file. All such files are called **Shell Scripts**.

Shell script can be defined as: "Shell Script is series of command written in plain text file.

Shell scripts are also referred as shell programs or shell procedures. The instructions stored in these files are executed in the interpretive mode – like batch file (.BAT) of Windows.

Following shell script has 2 commands stored in a file script1.sh. We can create the file with vi or emacs.

script1.sh	**echo "Hello"** **echo "Today is" `date`**

The extension .sh is used only for identification purpose. Shell scripts can have any extension or even we can have shell scripts without extension.

Executable permission is necessary to run any shell script. By default, a file does not have execution permission on creation. So to run shell script :

1. Use sh command as **sh script1.sh** at command prompt.

   OR

2. Use chmod command to assign executable status (right) to the file before executing it. And then run the script by typing the script name at prompt.
   e.g.
   ```
   $ chmod u+x script1.sh
   $ script1.sh
   ```
   First command will assign x permission means execution permission to u means user for file script1.sh. And next command will run the script.

Script executes the 2 statements in sequence. We are using shell as an interpreter, it is also a programming language. We can use all standard constructs like if, while and for in a shell script.

We can design shell scripts which can accept input at run time. There are two main approaches to design such scripts.

1. Write an interactive script that requests the input from the user.

   Input can be taken from user by appropriate message and then read the user's answer. We can use echo and read statement to prompt for input and reading the answer respectively.

   This approach is useful for creating programs to be used by occasional or inexperienced user.

2. Script can accept arguments from command line. These arguments come directly after the command.

   e.g. **cat script1.sh** command has script1.sh as a command line argument. It will tell the cat command to process script1.sh file.

   This approach is more powerful and flexible compared to first approach.

Example of interactive script

```
echo ‘Enter your name : “
read name
echo “Hello” $name “How are you?”
```

Why to Write Shell Script ?

1. Shell script can take input from user, file and output them on screen.
2. Useful to create our own commands.
3. Save lots of time.
4. To automate some task of day today life.
5. System Administration part can be also automated

FUNDAMENTAL SHELL PROGRAMMING CONSTRUCTS

CONDITIONAL EXECUTION

Instructions are executed sequentially in programs using sequence control structures. But sometimes we need to write program to do one thing under some situation and do something different in another situation.

For this, it is necessary to have control over the order of execution of the commands in the program. Following constructs offers this facility.

1. **if, if..else and nested if**

if condition is used for decision making in shell script. It takes two way decisions depending on fulfillment of certain condition.

There are 4 forms of if.

1) Simple if
2) if…else… fi
3) Nested if
4) if…elif

Syntax

1) Simple if	**2) if…else… fi**

if control command
then
    executable commands
fi
	if control command
	then
	    executable commands
	else
	    executable commands
	fi

if without else simplest form of if statement.

Group of commands between **then** and **else** is called an **if block** (or true block). Command between the **else** and **fi** is called **else block** (or false block).

Control command can be **condition or any Unix command**.

Condition can be formed with values, variables, logical operators and relational operators. Comparison between values will return true or false. Every Unix command return a value.

How if works

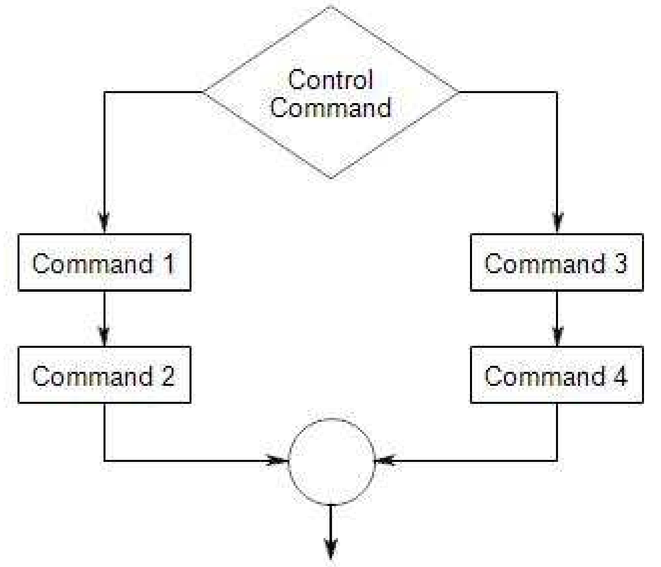
if statement of Unix is concerned with the exit status of a command. Exit status indicates whether the command was executed successfully or not. The exit status of a command is 0 if it is executed successfully. Otherwise it is 1.

if requires a **then**. It evaluates the success or failure of the command specified as control command. If the command executed successfully, the sequence of command following it is executed.

If the command fails, then the **else** statement is executed (if we have used else). else is optional.

Every if is closed with a corresponding fi.

Block diagram of if..else



Example of if… fi

```
#Script to compare two values
a=7
b=5
if [ $a –gt $b ]
then
echo “a is greater than b"
fi
```

Example of if…else…fi

```
#Script to copy file

if  cp $script1 $script2
then
     echo “File successfully copied"
else
     echo “Failed to copy"
fi
```

3) Nested if

```
if control command
then
      if control command
      then
             executable commands
     else ……
            executable commands
     fi
else ……………
      executable commands
fi
```

4) if… elif

```
if control command
then
      executable commands
elif control command
then
      ………….
else
     executable commands
fi
```

We can write entire if..else..fi construct within either the body of the if statement or the body of an else statement. This is called **nesting of ifs**.

In syntax, we have nested second if..else construct in first if statement. If the control command in first if executed successfully then control command in second if statement will be tested.

If it is true then commands in second if will be executed otherwise commands in else part of second if will be executed.

If the control command of first if fails then commands of else part of first if will be executed.

if…elif structure represents multiway branching. We can have many **eilf**s. The else part is optioal here.

If we use if…elif structure then only one fi is enough to close if. Nesting of if requires fi for every if.

if…elif structure allows us to group several alternatives one after another with the help of elif keyword.

Example of nested if

```
#Script to compare three values
a=5
b=7
c=9
if [ $a –gt $b –a $a –gt $b ]
then
    echo “a is largest"
else
    if [ $b –gt $a –a $b –gt $c ]
    then
          echo “b is largest"
    else
         echo “c is largest"
    fi
fi
```

Example of if…elif

```
#Script to compare three values
a=5
b=7
c=9
if [ $a –gt $b –a $a –gt $b ]
then
    echo “a is largest"
elif [ $b –gt $a –a $b –gt $c ]
then
      echo “b is largest"
 else
      echo “c is largest"
fi
```

test (companion of if)

When we use if condition to evaluate expressions, then **test** statement can be used as its control command.

test use some operators to evaluate the condition on its right. And it returns either true or false exit status which can be used by if for making decisions.

Syntax: test expression OR [expression]

Shorthand for test

A pair of rectangular brackets enclosing the expression can be used in place of test. So following two statements are equivalent:

```
test $x –eq $y
[  $x –eq $y ]
```

The [and] must have spaces on their inner sides. This form is easier.

test or [expr] works with

1. Numerical test - to compare two numbers
2. String tests - compare two strings / one string for null value
3. File tests – check file attributes

1. Numerical test

Numerical comparison operators used by test have a different form. They always begin with a – (hyphen) followed by two character word. Also it should be enclosed by whitespace on both side. For eg. –ne.

The operators are quite mnemonic (symbolic). –eq means equal to, -gt means greater than and so on.

- Numeric comparison in the shell is restricted to integer values only, decimal values are simply truncated.

For Numerical tests, use following operator in Shell Script

Operator	**Meaning**	**test statement with if command**	**[expr] statement with if command**
-eq	is equal to	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	if test 5 -ne 6	if [5 -ne 6]
-lt	is less than	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	if test 5 -ge 6	if [5 -ge 6]

2. String tests

test can be used to compare strings with another set of operators. Equality is performed with = and non equality is performed with !=.

These operators also should have whitespace on both the side.

For String tests use following operator in Shell Script

Operator	**Meaning**
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL
-n string1	Length of string1 is greater than 0
-z string1	Length of string1 is 0

- A test is negated by ! (bang) operator. test also allows checking of more than one condition using –a (AND) and –o (OR) operator.

3. File tests

test can be used to test various file attributes. For example, we can test whether a file has necessary read, write or executable permissions.

For File test use following operator in Shell Script

Test	**Meaning**
-s file	Non empty file
-f file	file exist and is a regular file
-d file	file exists and is a directory
-w file	file is writeable file
-r file	file is read-only file
-x file	file is executable file

2. <u>case</u>

The case statement is the second conditional statement offered by the shell. The statement exactly matches an expression for more than one alternative.

The case statement is a compact construct which allows multiway branching. It matches strings with wild cards which helps for string matching.

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

Syntax:

```
case  expression  in
   pattern1)   commands1  ;;
   pattern2)   commands2  ;;
   patternN)   commandsN ;;
   *)          commands   ;;
esac
```

case first matches expression with pattern1. If the match is successful then it executes commands1 which may be one or more commands.

If the match fails then pattern2 is matched and so on. Each command list is terminated with a pair of semicolon. Entire construct is closed with **esac** (reverse of **case**).

Expression will be compared against the patterns until a match is found. The default is *) and its executed if no match is found. *) is optional.

Example:

```
# to display car rent based on the car name supplied as command line argument
# Script name: car

rental=$1
case $rental in
     "car")       echo "For $rental Rs.20 per k/m";;
    "van")       echo "For $rental Rs.10 per k/m";;
    "jeep")      echo "For $rental Rs.5 per k/m";;
    "bicycle")   echo "For $rental 20 paisa per k/m";;
     *)          echo "Sorry, I can not get a $rental for you";;
esac
```

$ chmod +x car

$ car van	Output : For van Rs.10 per k/m
$ car car	Output : For car Rs.20 per k/m
$ car Maruti-800	Output : Sorry, I can not get a Maruti-800 for you

Matching multiple patterns

case can also match multiple patterns. We can use | (pipe) as the pattern delimiter when matching multiple patterns.

For e.g.

```
# to display car rent based on the car name supplied as command line argument
# Script name: car

rental=$1
case $rental in
     "car"|"van")             echo "For $rental Rs.20 per k/m";;
     "bicycle"|"tricycle")   echo "For $rental 20 paisa per k/m";;
     *)                      echo "Sorry, I can not get a $rental for you";;
esac
```

In this example, it will display same message for car or van. And same message for bicycle or tricycle.

Wild cards

case can use metacharacters *, ? and character class. Character class uses two or more characters represented by a pair of brackets [ ]. We can have multiple characters inside the bracket but matching takes place only for one character.

e.g.

```
case $string in
???*)   echo “string has more than 3 characters” ;;
A*)     echo “string starts with A” ;;
[Bb]*)  echo “string starts with B or b” ;;
esac
```

LOOPS

Loop is defined as: " A group of instruction that is executed repeatedly is called a loop."

Shell supports 3 types of loop:

1. for loop
2. while loop
3. until loop

Note that in each and every loop,

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

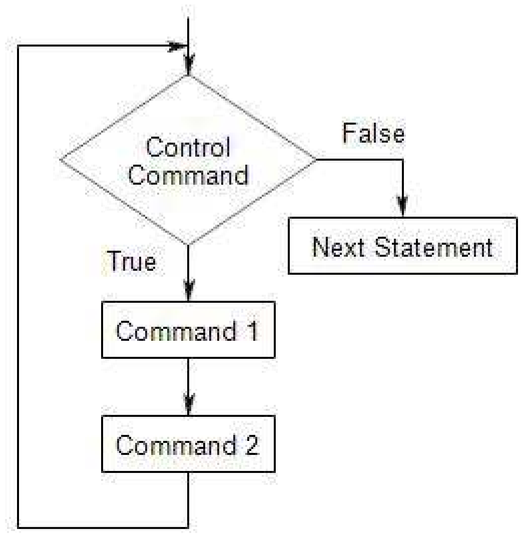
1. while

Syntax:

```
while [ condition ]
do
    commands

done
```

Block digram of while loop



<u>How while loop works?</u>

The while statement is very familiar to most programmers. It repeatedly performs a set of instructions till the control command returns a true exit status.

Commands enclosed by **do** and **done** are executed repeatedly as long as the condition remains true. We can use any Unix command or test as the control command.

For eg.

```
#Script to print first n natural numbers
#Script name : whiletest.sh
echo “Enter value of n : “
read n
i=1
while [ $i -le $n ]
do
  echo $i
  i=`expr $i + 1`
done
```

Above loop can be explained as follows:

read n	Takes values of n as an input from user
i=1	Set variable i to 1
while [$i -le $n]	This is our loop condition, here if value of i is less than n then, shell execute all statements between do and done
do	Start loop
echo $i	Print natural number as 1 2 .... n
i=`expr $i + 1`	Increment i by 1 and store result to i. (i.e. i=i+1) **<u>Caution:</u>** If you ignore (remove) this statement than our loop become infinite loop because value of variable i always remain less than n and program will only output 1 2 …. E (infinite times)
done	Loop stops here if i is not less than or equal to n i.e. condition of loop is not true. Henceloop is terminated.

<u>Note</u> until statement is complement of while construct. Body of the until loop will be executed repeatedly as long as the condition remains false.

2. <u>for</u>

for loop is more frequently used as compared to while and until loops.

for loop can be used :

- to execute a loop for a list of values or
- to execute a loop for fixed number of times

for loop has two formats.

1. for with list
2. for loop like C language

1. for with list

Syntax:

```
for variable in list
do
        commands
done
```

Here for loop will execute the sequence of commands for the values which are specified in the list after keyword **in**.

For e.g.

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

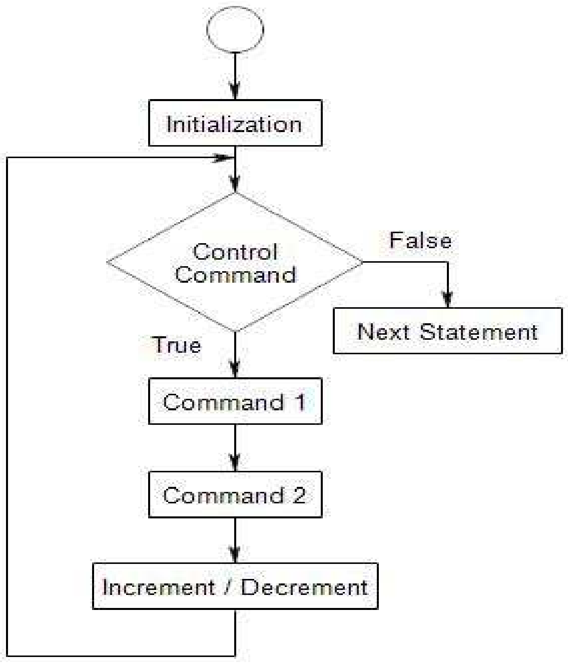
1) for loop first creates i variable.
2) Assign a number from the list of number 1 to 5 to i,.
3) The shell execute echo statement for each assignment of i. (This is usually known as iteration).
4) This process will continue until all the items in the list were not finished, because of this it will repeat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

2. for loop like C language

Syntax:

```
for (( expr1; expr2; expr3 ))
do
    commands
done
```

Block digram of for loop



<u>How for loop works?</u>

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 ((  i = 1 ;  i <= 5;  i++  ))
do
  echo "Welcome $i times"
done
```

```
Output:
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times
```

In above example, first expression (i = 1), 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 equel to 5) is TRUE.

Last expression **i++** increments the value of **i** by 1 i.e. it's equivalent to i = i + 1 statement.

<u>Nesting of for Loop</u>

Loop statement can be nested. To understand the nesting of for loop see the following shell script.

```
$ vi nestedfor.sh
for (( i = 1; i <= 5; i++ ))      ### Outer for loop ###
do

   for (( j = 1 ; j <= 5; j++ )) ### Inner for loop ###
   do
        echo -n "$i "
   done

  echo "" #### print the new line ###

done
```

```
Output:
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

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

<u>INPUT AND OUTPUT</u>

1. **<u>read</u>**

The read statement is the shell’s internal tool for taking input from user). read statement will make the scripts interactive.

It is used with one or more variables. Input given through the standard input (usually keyboard) is read into these variables.

When we use a statement like
read name
the script pauses at that point to take input from the keyboard. Whatever you enter will be stored in the variable **name**.

This is a form of assignment, so $ sign is not used before variable **name**.

Syntax: read variable1, variable2,...variableN

Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable fname.

#Script to read your name from key-board # echo "Your first name please:" read fname echo "Hello $fname, Lets be friend!"	Output: Your first name please: **bharat** Hello bharat, Lets be friend!

Note:

1) If the number of arguments supplied is less than number of variables in read then leftover variables will simply remain unassigned.
2) When number of arguments is more than number of variables in read then remaining words will be assigned to last variable.

2. printf

When writing a bash scripts most of us use echo command to print to standard output stream.

echo is easy to use and mostly it satisfies our needs without any problem. However simplicity very often comes limitation. This is also the case with echo command.
Formatting an echo command output can be a nightmare (terribale) and very often impossible task to do.

The solution to this can be the “printf” tool - a good old friend of all C/C++.

printf can be easily implemented into a script.

Syntax
printf accepts a **FORMAT string** and **arguments** in a following general form:
printf

In format string, prinft can have format specifiers, escape sequences or ordinary characters.

Arguments is the text that we want to print to standard output stream.

e.g.

printf “hello world”	Output: hello world$

We have supplied argument “hello world”. Not the different behaviour in comparison to echo command. No new line had been printed out. New line printing is default setting of echo command.

To print a new line we need to use escape sequence \n (new line) with format string in printf:

printf “hello world\n” OR printf "%s\n" "hello world"	Output: hello world $

The format string is applied to each argument:

$ printf "%s\n" "hello world" "in" "script"	Output: hello world in script

Format specifiers

The most commonly used printf specifiers are %s, %d and %f. The specifiers are replaced by corresponding arguments. See the following example:

To print an integer we can use %d specifier. %d specifiers refuses to print anything than integers.

To printf floating point numbers a %f specifier is used. The default behaviour of %f printf specifier is to print floating point numbers with 6 decimal places. To limit decimal places to 1 we can specify a precision as %.1f. Formatting to three places with preceding with 0, we can use %03d.

3. echo

Use echo command to display text or value of variable. echo is a built-in command which writes its arguments to standard output.

Standard output is the display screen by default, but it can be redirected to a file, printer, etc.

echo [options] [string, variables...]

Options

-n	Do not output the trailing new line.
\c	suppress trailing new line
\n	new line
\t	horizontal tab
\v	Vertical tab
\	backslash

The items in square brackets are optional. A string is any finite sequence of characters (i.e., letters, numerals, symbols and punctuation marks).

When echo is used without any options or strings, echo returns a blank line on the display screen followed by the command prompt on the subsequent line. This is because pressing the ENTER key is a signal to the system to start a new line, and thus echo repeats this signal.

When one or more strings are provided as arguments, echo by default repeats those stings on the screen.

Thus, for example, typing in the following and pressing the ENTER key would cause echo to repeat the phrase This is a pen. on the screen:

```
echo This is a pen.
```

It is not necessary to surround the strings with quotes, because it does not affect what is written on the screen. If quotes (either single or double) are used, they are not printed on the screen.

echo can do more than just repeat exactly what is written after it. It can also show the value of a particular variable.

Name of the variable will be preceded directly by the dollar character ($). $ will tells the shell to substitute the value of the variable. There should be no space between $ and variable name.
For example, a variable named x can be created and its value set to 5 with the following command:

```
x=5
```

The value of x can be recalled by the following:

```
echo The number is $x.
```

Echo is also useful for showing the values of environmental variables.

For example, to see the value of environmental variable HOME (current user's home directory), the following would be used:
echo $HOME

echo, by default, follows any output with a newline character. This is a non-printing (i.e., invisible) character that represents the end of one line of text and the start of the next line.

The –n option is used to suppress the default behaviour of echo to output with a newline character.

echo can likewise be a convenient way of appending text to the end of a file by using it together with the the append operator, which is represented by two consecutive rightward pointing angle brackets.

echo is also commonly used to have a shell script display a message or instructions, such as Please enter Y or N in an interactive session with users.

<u>TURNING DEBUGGING ON AND OFF</u>

Human beings are prone to mistakes. So the scripts may not work properly first time. If the grammar of script goes wrong then shell informs at the moment when incorrect statement gets executed.

Consider a case when the grammar is correct but there is a mistake in logic. The script will work but it may give you wrong results. At such times, we would like to debug the script.

We debug the script by tracing the flow of control. We want to check many things like – values of variables, substitution of command done properly or not etc.

To achieve this, we need to add following statement at the beginning of the script. This statement will turn **debugging on**.
set –vx

Here, **-v** ensures that each line in the script will be displayed before it gets executed. And **–x** ensures that command along with the argument values is also displayed before execution.

e.g.

set -vx echo Enter your name read name echo $name	echo Enter your name + echo Enter your name Enter your name read name + read name bharat mahan echo $name + echo bharat mahan bharat mahan

By setting –v option, the line which is about to be executed is displayed.

Lines which are preceded by + are displayed due to –x option. Note that variables have been substituted by their actual values in the lines preceded by +. This way we can come to know about value of variables on which a particular command is going to be operated.

To know which options have been set, we can use
$echo $- Output is let us say vx. It means that debugging is on.

This is an ideal tool. If we want to find that why our scripts are not working as per our expectation then this tool is very useful.

We can unset the debugging option (**debugging off**) by using:
set +vx

We can use set +vx at $ prompt or at the end of script in which it is turned on.

Disclaimer

The study material is compiled by Ami D. Trivedi. The basic objective of this material is to supplement teaching and discussion in the classroom in the subject. Students are required to go for extra reading in the subject through library work.