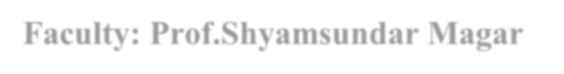
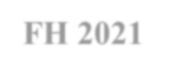
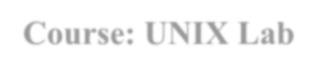
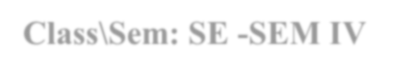
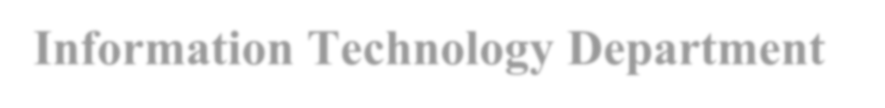
# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

**(PART A: TO BE REFFERED BY STUDENTS)**

**A.1 AIM:**

EXPERIMENT NO.08

**Study of Shell, Types of Shell, Variables and Operators**

**A.2 PREREQUISITE\REQUIREMENTS:**

1. Any Unix\Linux OS
2. Terminal (Bash Shell)

**A.3 OUTCOME:**

After successful completion of this experiment students will be able to

**LO2- Identify the UNIX general purpose commands**

**LO3-Apply Unix commands for system administrative tasks such as file system management and user management.**

**LO5- Implement basic shell scripts for different applications.**

**A.4 THEORY:**

SHELL

The shell is a program that takes your commands from the keyboard and gives them to the operating

system to perform. In the old days, it was the only user interface available on a UNIX computer. Nowadays, we have graphical user interfaces (GUIs) in addition to command line interfaces (CLIs) such as the shell.

On most Linux systems a program called bash (which stands for Bourne Again SHell, an enhanced version of the original Bourne shell program, sh, written by Steve Bourne) acts as the shell program. There are several additional shell programs available on a typical Linux system. These include: ksh, tcsh and zsh.

HOW TO DETERMINE SHELL

You can get the name of your shell prompt, with following command:

Syntax:

echo $SHELL

Look at the above snapshot, with the help of above command we got the name of our shell which is **'bash'.**

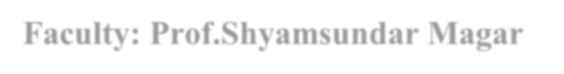
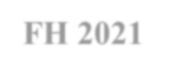
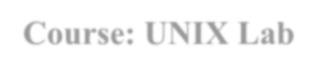
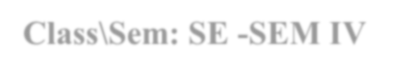
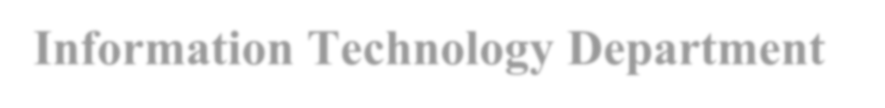
The $ sign stands for a shell variable, echo will return the text whatever you typed in.

SHELL SCRIPTING SHE-BANG

The sign #**!** is called she-bang and is written at top of the script. It passes instruction to program **/bin/sh.**

To run your script in a certain shell (shell should be supported by your system), start your script with #! followed by the shell name.

# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

**Example:**

#!/bin/bash

echo Hello World #!/bin/ksh

echo Hello World

A Shell provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output. Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

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. Following is a simple example of the date command, which displays the current date and time −

$date

Thu Jun 25 08:30:19 MST 2009

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

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 −

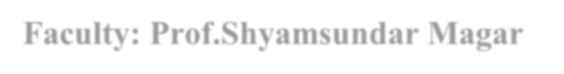
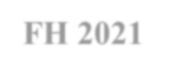
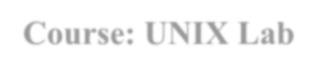
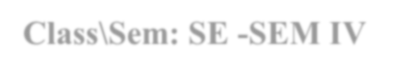
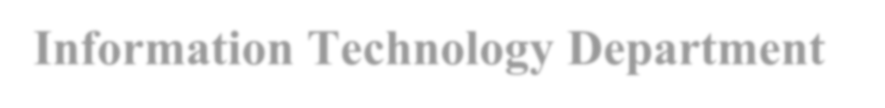
* Bourne shell (sh)
* Korn shell (ksh)
* Bourne Again shell (bash)
* POSIX shell (sh) The different C-type shells follow −
* C shell (csh)
* TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell". Bourne shell is usually installed as /bin/sh on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.

In this chapter, we are going to cover most of the Shell concepts that are based on the Borne Shell.

# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

SCRIPTS

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A

good shell script will have comments, preceded by # sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

We are going to write many scripts in the next sections. It would be a simple text file in which we would put all our commands and several other required constructs that tell the shell environment what to do and when to do it.

Shell scripts and functions are both interpreted. This means they are not compiled.

EXAMPLE SCRIPT

Assume we create a test.sh script. Note all the scripts would have the .sh extension. Before you add

anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example −

#!/bin/sh

This tells the system that the commands that follow are to be executed by the Bourne shell. It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.

To create a script containing these commands, you put the shebang line first and then add the commands

−

#!/bin/bash pwd

ls

SHELL COMMENTS

You can put your comments in your script as follows −

#!/bin/bash

# Script follows here:

pwd ls

Save the above content and make the script executable −

$chmod +x test.sh

The shell script is now ready to be executed −

$./test.sh

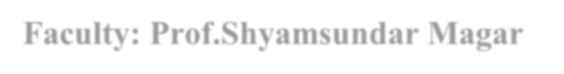
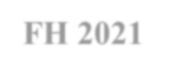
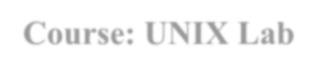
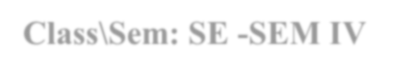
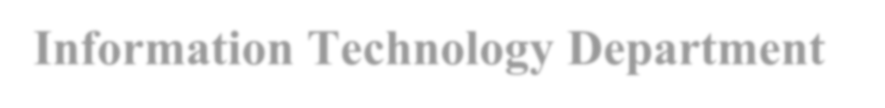
Upon execution, you will receive the following result −

/home/amrood

index.htm unix-basic\_utilities.htm unix-directories.htm test.sh unix-communication.htm unix-environment.htm

Note − To execute a program available in the current directory, use ./program\_name

# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

EXTENDED SHELL SCRIPTS

Shell scripts have several required constructs that tell the shell environment what to do and when to do

it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially. The following script uses the read command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

#!/bin/sh

# Script follows here:

echo "What is your name?" read PERSON

echo "Hello, $PERSON"

Here is a sample run of the script −

$./test.sh

What is your name? Shyamsundar

Hello, Shyamsundar

$

In this chapter, we will learn how to use Shell variables in Unix. 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

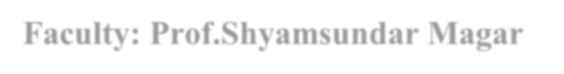
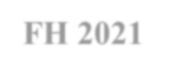
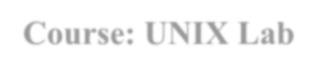
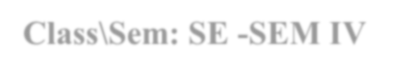
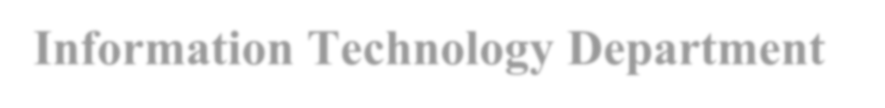
character ( \_). By convention, Unix shell variables will have their names in UPPERCASE. The following examples are valid variable names −

\_Shyam TOKEN\_A VAR\_1 VAR\_2

Following are the examples of invalid variable names − 2\_VAR

-VARIABLE VAR1-VAR2 VAR\_A!

# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

The reason you cannot use other characters such as !, \*, or - is that these characters have a special meaning for the shell.

**Defining Variables**

Variables are defined as follows − variable\_name=variable\_value For example − NAME="Shyamsundar"

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. Shell enables you to store any value you want in a variable. For example −

VAR1="Shyamsundar" VAR2=100

**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="Shyamsundar" echo $NAME

The above script will produce the following value − Shyamsundar

**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="Shyamsundar" readonly NAME NAME="Amit"

The above script will generate the following result − /bin/sh: NAME: This variable is read only.

**Unsetting Variables**

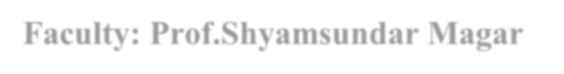
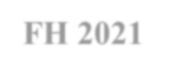
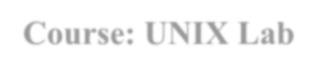
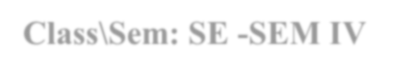
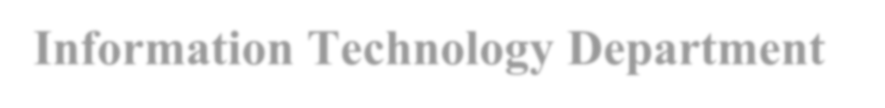
Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable. Following is the syntax to unset a defined variable using the unset command − unset variable\_name

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

#!/bin/sh NAME="Shyamsundar" unset NAME

echo $NAME

# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

The above example does not print anything. You cannot use the unset command to unset variables that are marked readonly.

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.
* we understood how to be careful when we use certain nonalphanumeric characters in variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.

For example, the $ character represents the process ID number, or PID, of the current shell −

* $echo $$

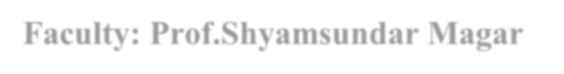
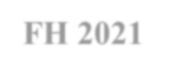
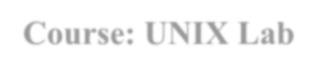
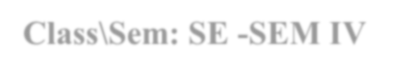
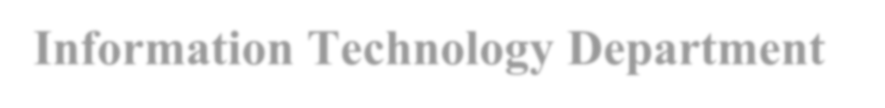
The above command writes the PID of the current shell −

29949

The following table shows a number of special variables that you can use in your shell scripts

|  |  |
| --- | --- |
| **S.N** | **Variable & Description** |
| 1 | **$0** -The filename of the current script. |
| 2 | **$n** -These variables correspond to the arguments with which a script was invoked. Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| 3 | **$#** -The number of arguments supplied to a script. |
| 4 | **$\*** -All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| 5 | **$@** -All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1  $2. |
| 6 | **$?** -The exit status of the last command executed. |

# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

|  |  |
| --- | --- |
| 7 | **$$** -The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| 8 | **$!** -The process number of the last background command. |

COMMAND-LINE ARGUMENTS

The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command. Following script uses various special variables related to the command line −

Here is a sample run for the above script −

#!/bin/sh

echo "File Name: $0"

echo "First Parameter : $1" echo "Second Parameter : $2" echo "Quoted Values: $@" echo "Quoted Values: $\*"

echo "Total Number of Parameters : $#"

$./test.sh Shyamsundar Magar File Name : ./test.sh

First Parameter : Shyamsundar Second Parameter : Magar

Quoted Values: Shyamsundar Magar Quoted Values: Shyamsundar Magar Total Number of Parameters : 2 applicable while naming arrays.

DEFINING ARRAY VALUES

The difference between an array variable and a scalar variable can be explained as follows. Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −

NAME01="Shyamsundar" NAME02="Amit"

NAME03="Abhishek" NAME04="Suresh" NAME05="John"

We can use a single array to store all the above mentioned names. Following is the simplest

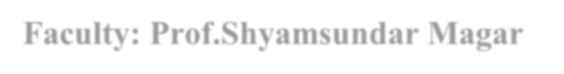
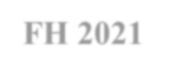
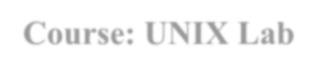
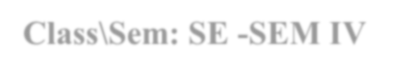
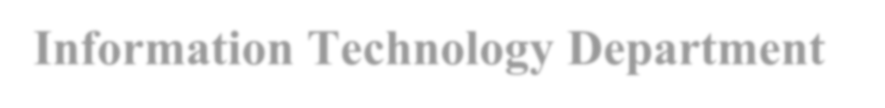
method of creating an array variable. This helps assign a value to one of its indices.

array\_name[index]=value

Here *array\_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands −

# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

NAME[0]=" Shyamsundar " NAME[1]=" Amit "

NAME[2]=" Abhishek " NAME[3]="Suresh" NAME[4]="John"

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

set -A array\_name value1 value2 ... valuen

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

array\_name=(value1 ... valuen)

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. Following is an example to understand the concept −

#!/bin/sh NAME[0]="Shyamsundar" NAME[1]="Amit"

NAME[2]="Abhishek" NAME[3]="Suresh" NAME[4]="John"

echo "First Index: ${NAME[0]}" echo "Second Index: ${NAME[1]}"

The above example will generate the following result −

$./test.sh

First Index: Shyamsundar Second Index: Amit

You can access all the items in an array in one of the following ways −

${array\_name[\*]}

${array\_name[@]}

Here **array\_name** is the name of the array you are interested in. Following example will help you understand the concept −

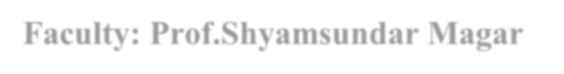
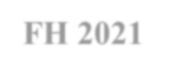
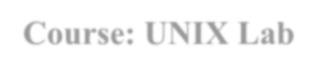
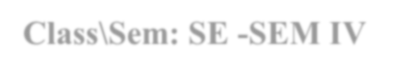
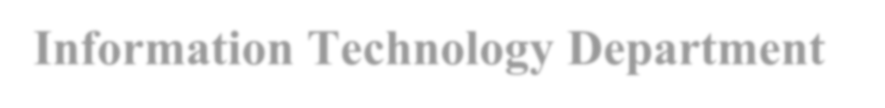
#!/bin/sh NAME[0]="Shyamsundar" NAME[1]="Amit"

NAME[2]="Abhishek" NAME[3]="Suresh" NAME[4]="John"

echo "First Method: ${NAME[\*]}" echo "Second Method: ${NAME[@]}"

The above example will generate the following result −

# Terna Engineering College Information Technology Department



**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

$./test.sh

First Method: Shyamsundar Amit Abhishek Suresh John Second Method: Shyamsundar Amit Abhishek Suresh John

OPERATORS

There are various operators supported by each shell. We will discuss in detail about Bourne

shell (default shell) in this chapter. We will now discuss the following operators −

* + Arithmetic Operators
  + Relational Operators
  + Boolean Operators
  + String Operators
  + File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**. The following example shows how to add two numbers −

#!/bin/sh val=`expr 2 + 2`

echo "Total value : $val"

The above script will generate the following result −

Total value : 4

The following points need to be considered while adding −

* + There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
  + The complete expression should be enclosed between **‘ ‘**, called the backtick.

ARITHMETIC OPERATORS

The following arithmetic operators are supported by Bourne Shell. Assume variable **a** holds 10

and variable **b** holds 20 then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |

# Terna Engineering College Information Technology Department

**Class\Sem: SE -SEM IV Course: UNIX Lab**

**FH 2021**

**Faculty: Prof.Shyamsundar Magar**

**LAB Manual**

|  |  |  |
| --- | --- | --- |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[ $a == $b ]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

RELATIONAL OPERATORS

Bourne Shell supports the following relational operators that are specific to numeric values.

These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then −

**Operator**

**Description**

**Example**

**-eq**

Checks if the value of two operands are equal or not; if yes, then the condition becomes true.

[ $a -eq $b ] is not true.

**-ne**

Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.

[ $a -ne $b ] is true.

**-gt**

Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.

[ $a -gt $b ] is not true.

**-lt**

Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.

[ $a -lt $b ] is true.

**-ge**

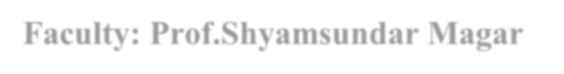
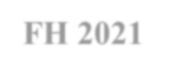
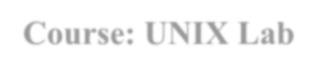
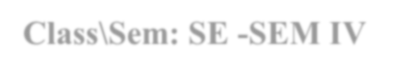
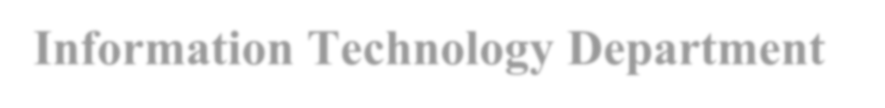
Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.

[ $a -ge $b ] is not true.

**-le**

Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.

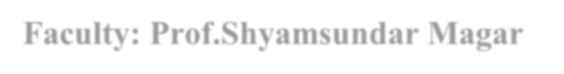
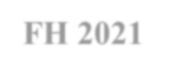
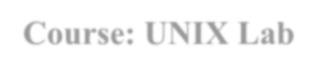
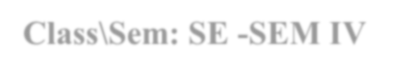
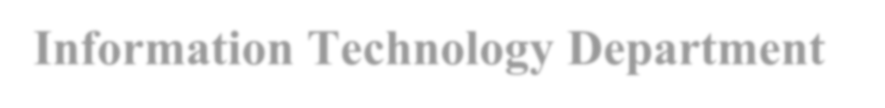
[ $a -le $b ] is true.



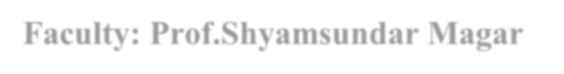
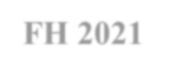
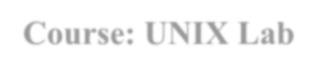
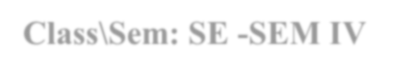
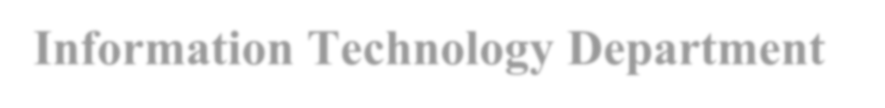
It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[ $a <= $b ]** is correct whereas, **[$a <=**

**$b]** is incorrect.

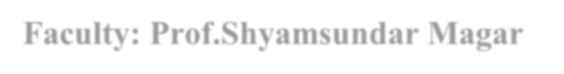
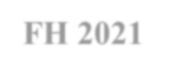
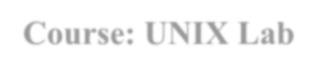
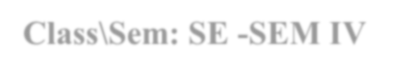
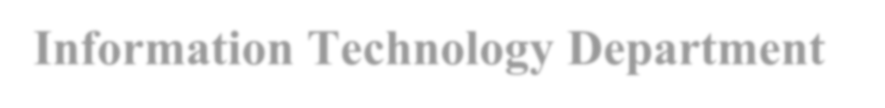
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Terna Engineering College Information Technology Department**  **Class\Sem: SE -SEM IV Course: UNIX Lab**  **FH 2021**  **Faculty: Prof.Shyamsundar Magar**  **LAB Manual**  BOOLEAN OPERATORS  The following Boolean operators are supported by the Bourne Shell. Assume variable **a** holds 10 and variable **b** holds 20 then − | | | | |
|  | **Operator** | **Description** | **Example** | |
| **!** | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. | |
| **-o** | This is logical **OR**. If one of the operands is true, then the condition becomes true. | [ $a -lt 20 -o $b -gt 100 ] is true. | |
| **-a** | This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false. | [ $a -lt 20 -a $b -gt 100 ] is false. | |
| STRING OPERATORS  The following string operators are supported by Bourne Shell. Assume variable **a** holds "abc" and variable **b** holds "efg" then − | | | | |
|  | **Operator** | **Description** | | **Example** |
| **=** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | | [ $a = $b ] is not true. |
| **!=** | Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true. | | [ $a != $b ] is true. |
| **-z** | Checks if the given string operand size is zero; if it is zero length, then it returns true. | | [ -z $a ] is not true. |
| **-n** | Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true. | | [ -n $a ] is not false. |
|  | | | | |



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Terna Engineering College Information Technology Department**  **Class\Sem: SE -SEM IV Course: UNIX Lab**  **FH 2021**  **Faculty: Prof.Shyamsundar Magar**  **LAB Manual** | | | | | |
|  | **str** | | Checks if **str** is not the empty string; if it is empty, then it returns false. | [ $a ] is not false. | |
| FILE TEST OPERATORS  We have a few operators that can be used to test various properties associated with a Unix file.  Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on − | | | | | |
|  | **Operator** | **Description** | | | **Example** |
| **-b file** | Checks if file is a block special file; if yes, then the condition becomes true. | | | [ -b $file ] is false. |
| **-c file** | Checks if file is a character special file; if yes, then the condition becomes true. | | | [ -c $file ] is false. |
| **-d file** | Checks if file is a directory; if yes, then the condition becomes true. | | | [ -d $file ] is not true. |
| **-f file** | Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true. | | | [ -f $file ] is true. |
| **-g file** | Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true. | | | [ -g $file ] is false. |
| **-k file** | Checks if file has its sticky bit set; if yes, then the condition becomes true. | | | [ -k $file ] is false. |
| **-p file** | Checks if file is a named pipe; if yes, then the condition becomes true. | | | [ -p $file ] is false. |
| **-t file** | Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true. | | | [ -t $file ] is false. |
| **-u file** | Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. | | | [ -u $file ] is false. |
| **-r file** | Checks if file is readable; if yes, then the condition becomes true. | | | [ -r $file ] is true. |
| **-w file** | Checks if file is writable; if yes, then the condition becomes true. | | | [ -w $file ] is true. |
| **-x file** | Checks if file is executable; if yes, then the condition becomes true. | | | [ -x $file ] is true. |
|  | | | | | |



|  |  |  |  |
| --- | --- | --- | --- |
| **Terna Engineering College Information Technology Department**  **Class\Sem: SE -SEM IV Course: UNIX Lab**  **FH 2021**  **Faculty: Prof.Shyamsundar Magar**  **LAB Manual** | | | |
|  | **-s file** | Checks if file has size greater than 0; if yes, then condition becomes true. | [ -s $file ] is true. |
| **-e file** | Checks if file exists; is true even if file is a directory but exists. | [ -e $file ] is true. |
| PART B  (PART B: TO BE COMPLETED BY STUDENTS)  ***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***  **Roll No. 100 Batch : A1**  **Name:** Shyamsundar Magar Date of Submission 09/01/2020  **Class :** SE-IT Date of Experiment: 16/01/2020   * 1. DOCUMENT CREATED BY THE STUDENT:   *Describe the shell variables and operators*   * 1. OBSERVATIONS AND LEARNING:   *(Students are expected to understand the selected topic. Write your observations after learning)*   1. *Write shell script to use Shell Variables* 2. *Write shell script to use arithmetic operators* 3. *Write shell script to use relational operators*   B.3 CONCLUSION:  ***(Students must write the conclusion as per the attainment of individual outcome listed above and learning/observation noted in section B.2)***  *Hence we studied and executed the Shell script variables and operators*.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | | | |



|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |