# OPERATING SYSTEMS LABORATORY MANUAL

**SHELL PROGRAMMING BASICS**

## AIM

To execute the fundamentals of shell programming such as control flow

statements.

### EXISTING PROBLEM

* It is not possible to perform more than one task at a time using shell command

### SHELL SCRIPTS (MULTITASKING)

* In order to solve the problems of shell command, the shell programming is introduced here
* Doing more than one job at a time (multitasking)
* It is also called as **shell programming**

### VARIABLES SECTION

* Names given to the memory location
* Shell program supports **dynamic data typed system** which means that no need to use specific data type for variables declaration

### Syntax

Variable-Name=Initial-Value

**Example**

a=10

str=”Sachin” c=’G’

f=true

readonly id=14

# integer type # string type

# character type # boolean value

# integer constant

### EXAMPLE OF DATA TYPES IN SHELL CODE SOURCE CODE

echo " " echo "\tShell Data Types"

echo " " # variables definition

a=19 b=15.45 c='S'

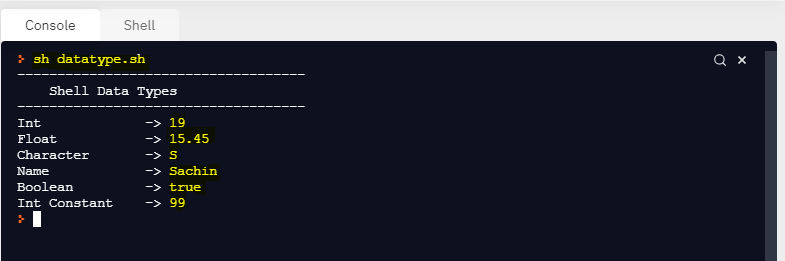
str="Sachin" flag=true

# constant variable definition readonly id=99

echo "Int\t\t\t\t-> $a" echo "Float\t\t\t-> $b" echo "Character\t\t-> $c" echo "Name\t\t\t-> $str"

echo "Boolean\t\t\t-> $flag" echo "Int Constant\t-> $id"

### OUTPUT



**COMMENT LINE STATEMENTS**

* + Usually these statements are ignored by compiler or interpreter
  + Like c/c++, shell supports two types of comment line statements. They are

1. Single line statement
2. Multi line statements

### Single line statement

* + The single line statement is indicated by **# symbol** in shell program

### Example

# This is single line statement

1. **Multi line statements**
   * It is used to ignore more than one statements
   * This is indicated by :’ ’ symbol in shell program

### Example

**:’**

Variable Declarations a=20

k=25

**‘**

**SELECTION STATEMENTS**

1. Simple If statement
2. If else Statement
3. If else if Statement
4. Case Statement

### Simple If Statement Syntax

if [ condition ] then

true statement

fi

* + It is an important to note that, the space should be given before and after the operator symbol [
  + You can use test keyword instead of the operator symbols [ ]

### If else Statement Syntax

if [ condition ] then

true statement

else

false statement

fi

1. **If…elif…else Statement**

Syntax

if [ condition ] then

true statement elif [ condition ]

then

true statement

else

false statement

fi

* + It is an important to note that, the simple if, if else and if-elif-else should be closed by fi keyword.

### Case Statement

* + It is equivalent to switch case statements in c language
  + It is used to execute several statements based on the value of expression
  + This is done by using the reserved word case
  + It is an alternative option for if..elif..else statements

**Syntax**

case <variable> in Pattern 1)

Commands / statements

;;

Pattern 2)

Commands / statements

;;

…

easc

**Where,**

;;  represents the break part in case statements

### LOOPING STATEMENTS

1. While loop (while)
2. Until loop (until)
3. For loop (for)

### While loop Syntax

while [ condition ] do

true statement

done

**Infinite While loop**

* + It is an important to note that, **the colon (:) operator or true keyword** is used for creating an infinite loop
  + The colon (:) operator is used instead of the operator symbols **[ ]**

### EXAMPLE OF INFINITE LOOP USING WHILE LOOP

**SOURCE CODE**

while :

do

echo "Hello World" done

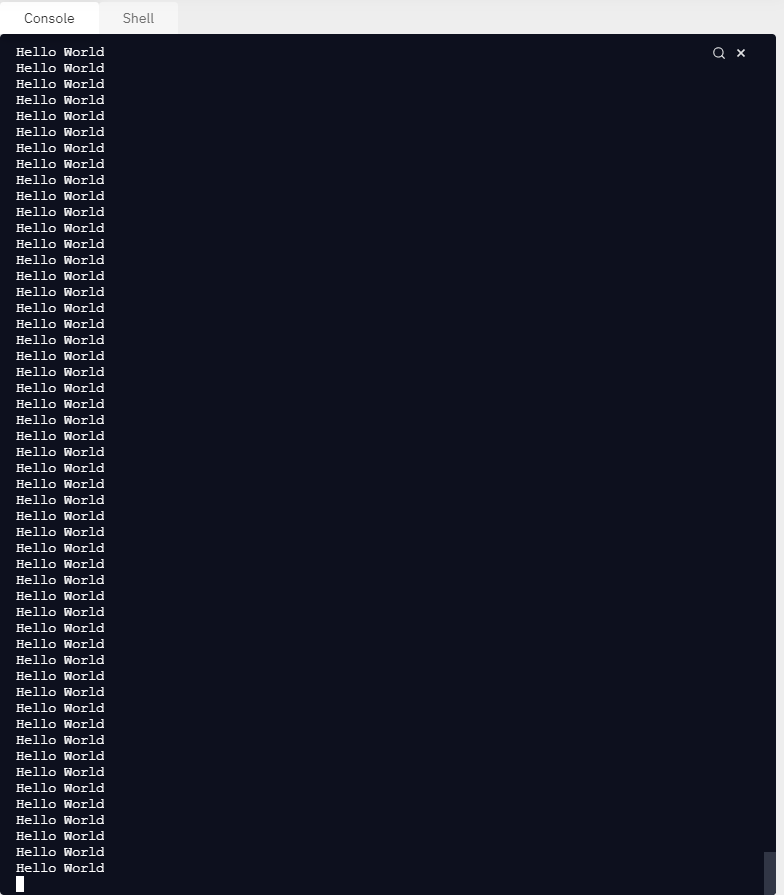
while true do

echo "Hello World"

(OR)

done

### OUTPUT



1. **Until loop**
   * Here loop is executing until the condition is false
   * If the condition becomes true it will exit from the loop

**Syntax**

until [ condition ] do

true statement

done

* + 1. **EXAMPLE OF UNTIL LOOP**

### SOURCE CODE

echo " " echo "\t\tUntil Loop Example"

echo " " i=1



Here the looping statements are executed until the condition becomes fail like **1>5, 2>5, 3>5, 4>5, 5>5**

until [ $i -gt 5 ] do

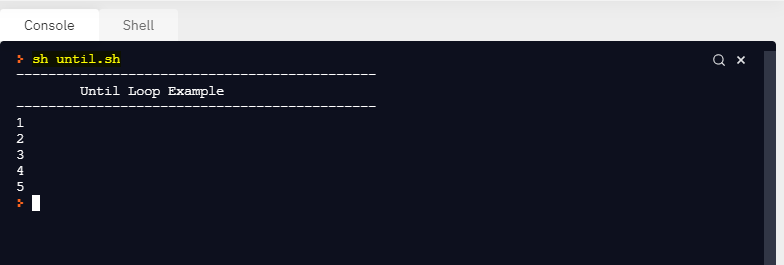
echo $i i=`expr $i + 1`

done



This loop will terminate whenever the condition becomes true like **6>5**

### OUTPUT



1. **For loop**

Syntax

for variable in w1 w2 … wn do

true statement

done

### Where,

Variable can be any user defined name

w1 w2 …wn  list of the values separated by spaces

### EXAMPLE OF FOR LOOP

**SOURCE CODE**

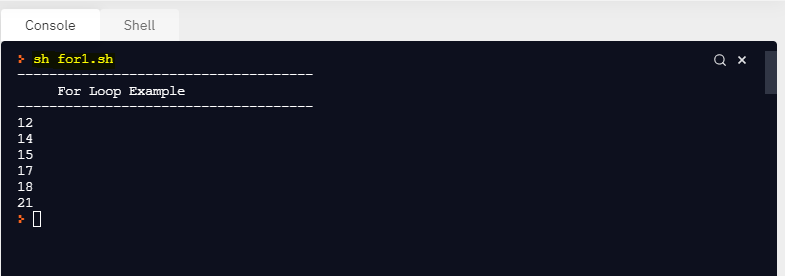
echo " " echo "\t For Loop Example"

echo " " for i in 12 14 15 17 18 21

do

echo $i done

### OUTPUT



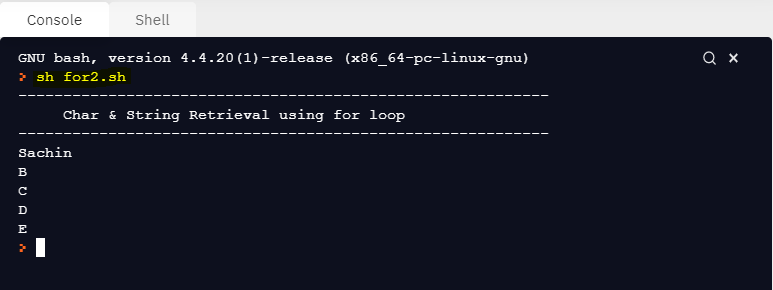
* + 1. **CHARACTERS AND STRING RETRIEVAL USING FOR LOOP SOURCE CODE**

echo " " echo "\t Char & String Retrieval using for loop" echo " " for i in 'Sachin' 'B' "C" "D" ‘E’

do

echo $i done

### OUTPUT



* + 1. **DISPLAYING FILES AND DIRECTORIES USING FOR LOOP SOURCE CODE**

echo " " echo "\t Listing Files and Folder using for loop"

echo " " # get all the files and store them to variable



fset=`ls` k=1

Store list of files to the variable fset using ls command.

# loop the variable fset for i in $fset

do

k=k+1 or k++

echo "$k. $i" k=`expr $k + 1`

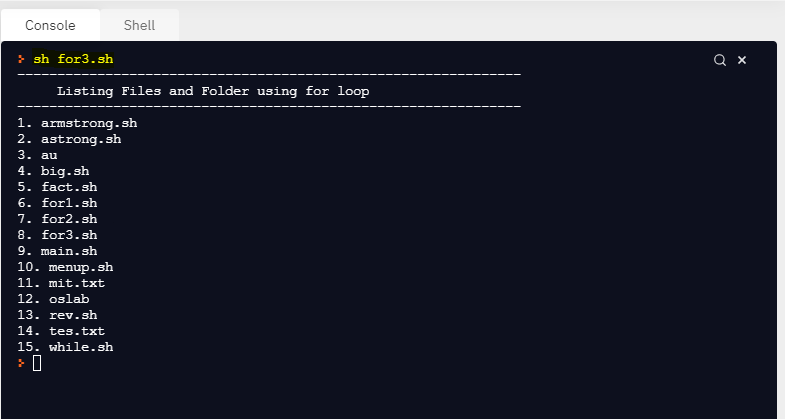
done



Command Substitution:

Storing the output of command to a user defined variable. This is done by using the operator `command` or $(command)

### OUTPUT



* + 1. **FACTORIAL OF A NUMBER**

Language : shell (.sh)

Editor : **replit.com (Online Linux Terminal)**

OS : Windows 10

### SOURCE CODE

echo " " echo "\t\tFactorial Program"

echo " " echo "Enter a number : "

read n i=0 f=1

while [ $i -lt $n ]

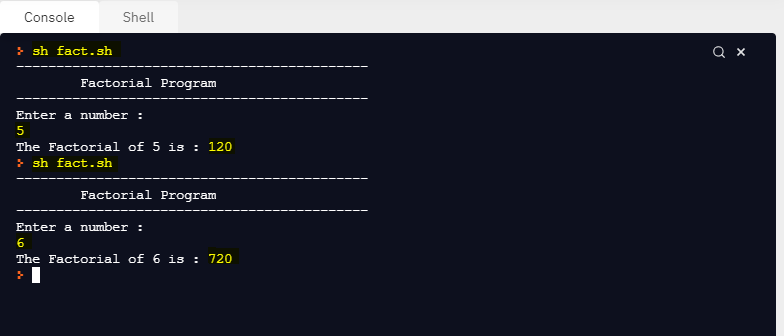
do

# increment i by 1 i=`expr $i + 1` f=`expr $f \\* $i`

done

echo "The Factorial of $n is : $f"

### OUTPUT



**C STYLE CODING IN SHELL**

* In shell, we can use c style coding for looping statements and expressions
* The expressions are defined by **$(exp)**. Here each is also closed by () symbol.

### FACTORIAL OF A NUMBER USING C-STYLE SOURCE CODE

echo " -" echo "\t\tFactorial Program- C Style"

echo " -" echo "Enter a number : "

read n i=1 f=1

# while loop while [ $i -le $n ] do

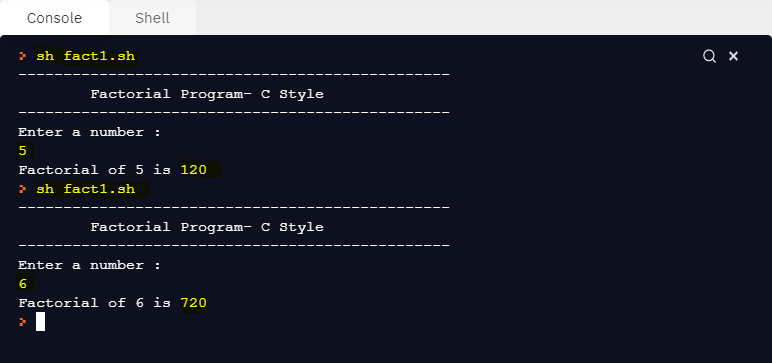
# expression 1 in c-style f=$((f\*i))

# expression 2 in c-style i=$((i+1))

done

echo "Factorial of $n is $f"

### OUTPUT



* + 1. **REVERSE OF A NUMBER**

### SOURCE CODE

echo " " echo "\t\tReverse of Number"

echo " " echo "Enter a number : "

read n duplicate=$n res=0

while [ $n -ne 0 ] do

# find the reminder rem=`expr $n % 10`

# multiply reverse number with 10 res=`expr $res \\* 10`

Read a number

while (n!=0)

rem=n%10

res=res\*10

# add the resultant number with remainder number res=`expr $res + $rem`

# divide n by 10 n=`expr $n / 10`

n=n/10

done

echo "The Reverse Number of $duplicate is $res"

### OUTPUT



* + 1. **ARMSTRONG NUMBER**

### SOURCE CODE

echo " " echo "\t\tArmstrong Number"

echo " " echo "enter a number"

read n

# variables declarations dup=$n



arm=0

# while loop

while test $n -ne 0 do

# find remainder of a number rem=`expr $n % 10`

# multiply rem with three times rem=`expr $rem \\* $rem \\* $rem`

# add the resultant remainder with arm variable

while (n!=0)

rem=n%10

rem=(rem\*rem\*rem)

arm=`expr $arm + $rem` # divide n by 10

n=`expr $n / 10` done

n=n/10

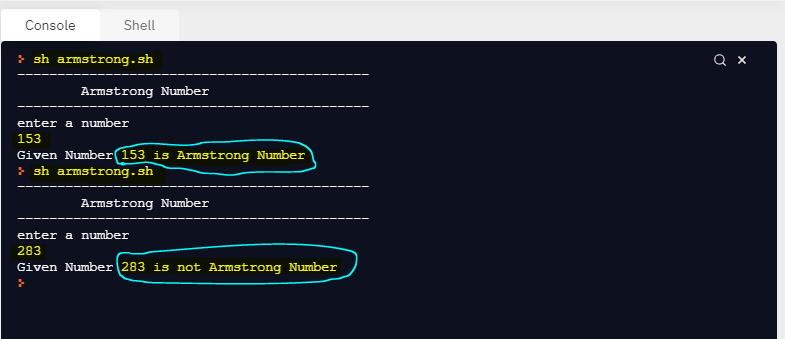
if [ $dup -eq $arm ] then

echo "Given Number $dup is Armstrong Number" else

echo "Given Number $dup is not Armstrong Number"

fi

### OUTPUT



* + 1. **MENU DRIVEN PROGRAM**

### SOURCE CODE

while [ true ] do

echo " " echo "\t\t Menu Program"

echo " " echo "1. View Files \t 2.Date"

echo "3. Users List \t4.Calendar" echo "5. Exit"

echo "\tEnter ur choice : " read ch

#switch case case $ch in

1. ls;;
2. date;;

3) w;;

1. cal;;
2. exit;; esac

# read choice from user for continuation of program execution echo "Do you want to continue : Press Yes/No"

read ch

if [ $ch = "yes" ] || [ $ch = "Yes" ] || [ $ch = "YES" ] then

continue else

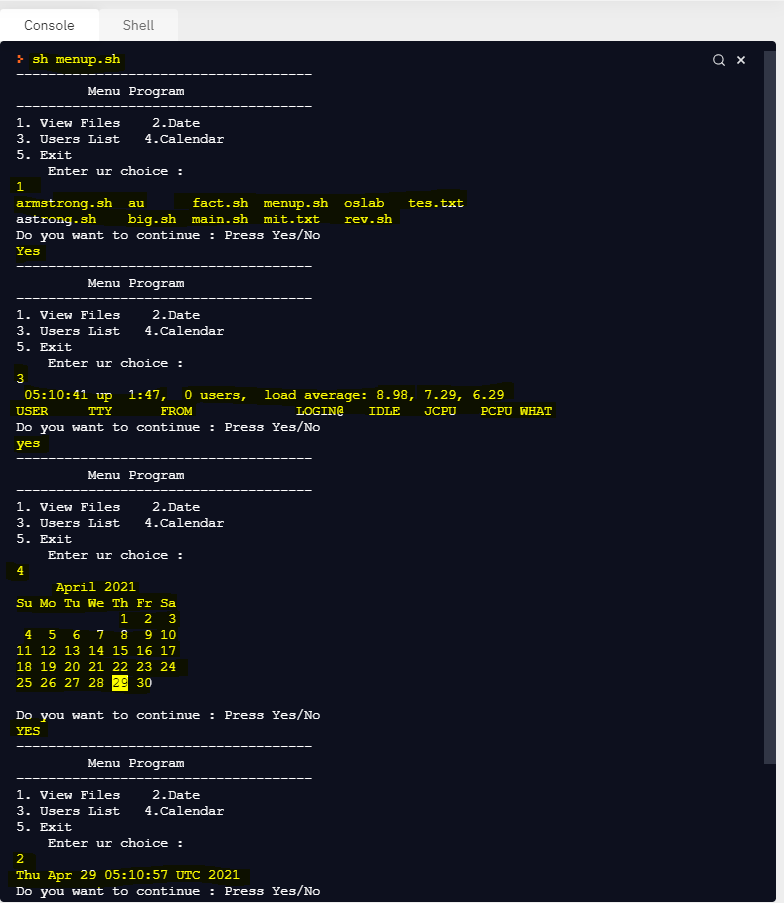
exit

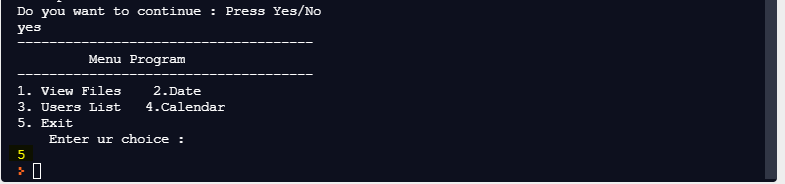
fi done

### NOTE

* It is an important to note that, the single equal operator (**=**) is used for **string comparison** and the symbol **–eq** or **==** is used for **number comparison** in the shell program.

### OUTPUT



**OUTPUT – (CON)**

### COMMAND LINE ARGUMENTS (POSITIONAL PARAMETERS)

* Process of passing the input arguments to the program at the time of execution is called as command line arguments
* In shell, this is done with help of $ symbol

|  |  |  |
| --- | --- | --- |
| **S.N** | **POSITIONAL PARAMETERS** | **DESCRIPTION** |
| 1. | $0 | Indicates the filename itself |
| 2. | $1 | Indicates the first argument |
| 3. | $2 | Indicates the second argument |
|  | … |  |
| 4. | $\* | Represents the total number of input arguments which are submitted to the shell program |
| 5. | $# | Shows the count of total number of arguments passed to the shell program |

### EXAMPLE OF COMMAND LINE INPUTS

**SOURCE CODE**

echo " " echo "\t\tCommand Line Arguments (Input)" echo " " a=$1

b=$2

# check whether the command line arguments are submitted or not if [ $# -ne 0 ]

This is equivalent to if [ $# != 0 ]

+ $b`

then

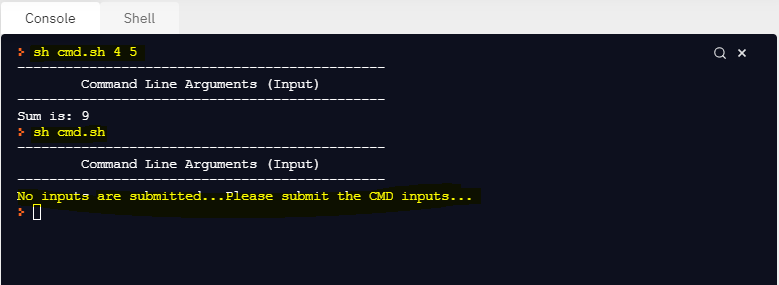
r=`expr $a

echo "Sum is: $r" else

echo "No inputs are submitted...Please submit the CMD inputs..."

fi

### OUTPUT



* 1. **EXAMPLE OF COMMAND LINE INPUTS – MUCH ARGUMENTS SOURCE CODE**

echo " " echo "\t\tCommand Line Arguments"

echo " "

# check whether the command line arguments are submitted or not if [ $# -ne 0 ]

then

echo "Total Input Arguments are Submitted: $#" for i in $\*

This is equivalent to if [ $# != 0 ]

do

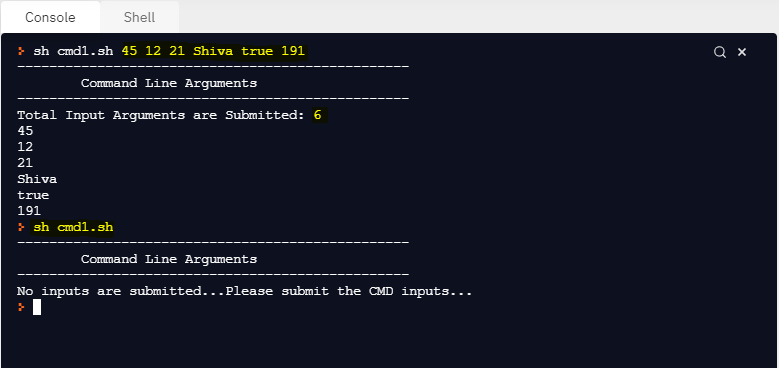
echo $i done

else

echo "No inputs are submitted...Please submit the CMD inputs..."

fi

### OUTPUT



**RESULT**

Thus the basics of shell programming was executed successfully.

## EX.NO: 4 SHELL SCRIPTING – OPERATORS, FUNCTIONS

### AIM

To practice the different types of operators using shell programming

### OPERATORS

* A special symbol which is used to perform the various tasks such as arithmetic operations, relational operations, logical operations, file testing operations, comparisons, etc, …

### ARITHMETIC OPERATORS

|  |  |  |
| --- | --- | --- |
| **S.N** | **OPERATOR** | **DESCRIPTION** |
| 1. | + | Addition |
| 2. | - | Subtraction |
| 3. | \* | Multiplication |
| 4. | / | Division |

**EXPRESSION IN SHELLS**

* Shell provides two options for performing expressions in shell scripts. They are
  + 1. Using expr command  Shell style expr
    2. Using double braces ()  C Style $(exp)

### NOTE

* It is an important to note that, the operator **\*** does not provide the multiplication in shell expression using expr command. Because in shell, the operator \* means wild card characters.
* So, the backward slash followed by \* symbol **\\*** is to provide the multiplication expression in shell expression using expr command.

### EXAMPLE OF ARITHMETIC OPERATIONS USING EXPR COMMAND SOURCE CODE

echo " " echo "\tArithmetic Operations using expr command" echo " " echo "Enter the number 1: "

read a

echo "Enter the number 2: " read b

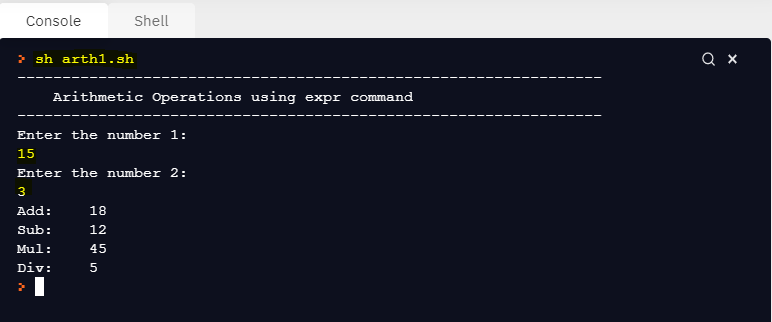
# performing arithmetic operations using expr command r1=`expr $a + $b`

r2=`expr $a - $b` r3=`expr $a \\* $b` r4=`expr $a / $b` # print the results echo "Add: \t$r1" echo "Sub: \t$r2" echo "Mul: \t$r3" echo "Div: \t$r4"



The usage of **\*** in expr command does not directly support for multiplication. Because it gives different meanings. So the **backward slash with \*** symbol provides the multiplication in the expr command.

### OUTPUT



1. **EXAMPLE OF ARITHMETIC OPERATIONS USING C STYLE SOURCE CODE**

echo " " echo "\tArithmetic Operations using C-Style"

echo " "



echo "Enter the number 1: " read a

echo "Enter the number 2: " read b

The Syntax is $(e1, e2,..en).

Each expression e1, e2, ..en can be closed by ()

# performing arithmetic operations using expr command r1=$((a+b))

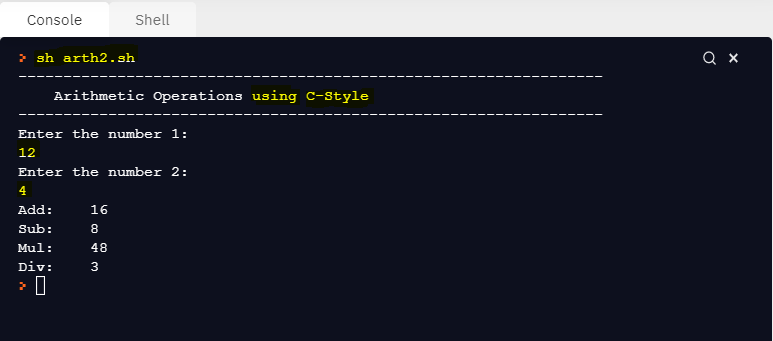
r2=$((a-b))

r3=$((a\*b))

r4=$((a/b))

# print the results echo "Add: \t$r1" echo "Sub: \t$r2" echo "Mul: \t$r3" echo "Div: \t$r4"

### OUTPUT



**RELATIONAL OPERATORS**

### RELATIONAL OPERATORS FOR NUMBERS [ using $() ]

|  |  |  |
| --- | --- | --- |
| **S.N** | **OPERATOR** | **DESCRIPTION** |
| 1. | == | Equal |
| 2. | != | Not Equal |
| 3. | < | Lesser than |
| 4. | <= | Lesser than or Equal to |
| 5. | > | Greater than |
| 6. | >= | Greater than or Equal to |

**NUMERIC COMPARISON OPERATORS FOR NUMBERS**

|  |  |  |
| --- | --- | --- |
| **S.N** | **OPERATOR** | **DESCRIPTION** |
| 1. | -eq | Equal |
| 2. | -ne | Not Equal |
| 3. | -gt | Greater than |
| 4. | -ge | Greater than or Equal to |
| 5. | -lt | Lesser than |
| 6. | -le | Lesser than or Equal to |

### RELATIONAL OPERATORS FOR STRINGS [ using if ]

|  |  |  |
| --- | --- | --- |
| **S.N** | **OPERATOR** | **DESCRIPTION** |
| 1. | = | Equal |
| 2. | != | Not Equal |
| 3. | \< | Lesser than |
| 4. | \> | Greater than |

1. **EXAMPLE OF RELATIONAL OPERATORS USING C STYLE SOURCE CODE**

echo " " echo "\tRelational Operators using C-Style"

echo " " echo "Enter the number 1: "

read a

echo "Enter the number 2: " read b

# performing relational operators using C style $() r1=$((a==b))

r2=$((a!=b))

r3=$((a<b))

r4=$((a<=b))

r5=$((a>b))

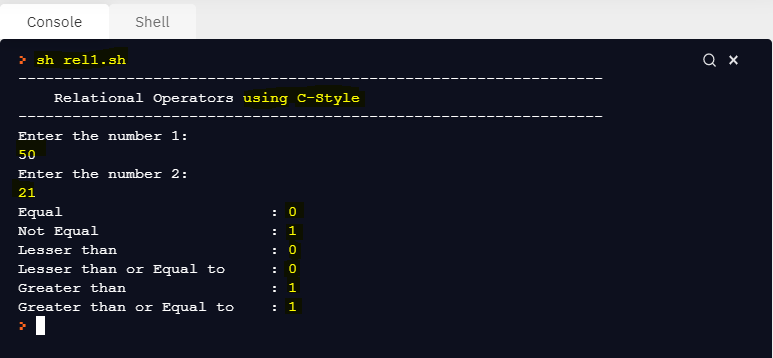
r6=$((a>=b))

# print the relational results echo "Equal \t\t\t\t\t\t: $r1" echo "Not Equal \t\t\t\t\t: $r2" echo "Lesser than \t\t\t\t: $r3"

echo "Lesser than or Equal to \t: $r4" echo "Greater than \t\t\t\t: $r5"

echo "Greater than or Equal to \t: $r6"

### OUTPUT



1. **EXAMPLE OF RELATIONAL OPERATORS FOR STRINGS SOURCE CODE**

echo " " echo "\tRelational Operators for Strings"

echo " " echo "Enter the name 1: "

read a

echo "Enter the name 2: " read b

# performing relational operators for strings if [ $a = $b ]

then

echo " Equal Results " echo "Both Strings are Equal"

else

echo " Equal Results " echo "Both Strings are NOT Equal"

fi

if [ $a != $b ]

then

echo "----------NOT Equal Results "

echo "Both Strings are NOT Equal" else

echo "----------NOT Equal Results "

echo "Both Strings are Equal"

fi

if [ $a \> $b ] then

echo "----------Greater Results "

echo "$a is Greater than $b" else

echo "----------Greater Results "

echo "$b is Greater than $a"

fi

if [ $a \< $b ] then

echo "----------Lesser Results "

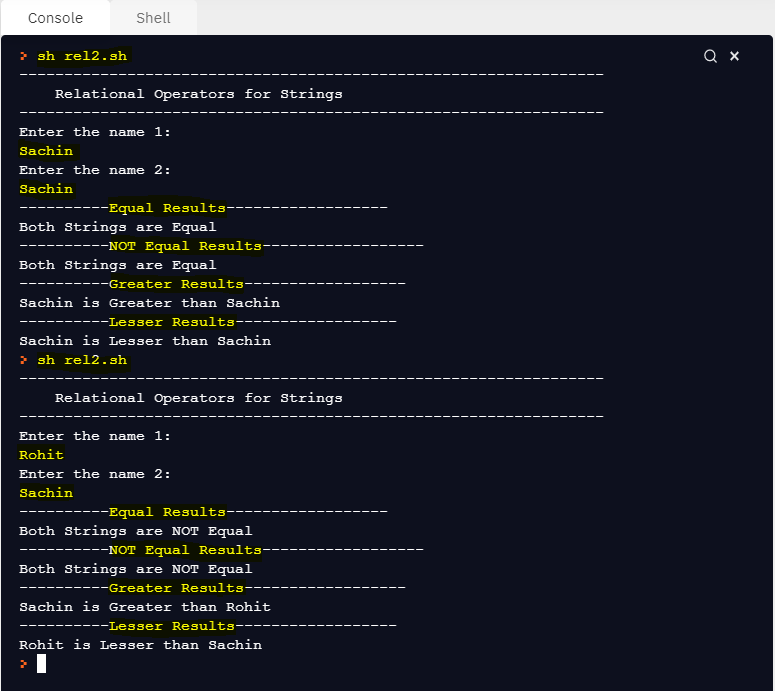
echo "$a is Lesser than $b" else

echo "----------Lesser Results "

echo "$b is Lesser than $a"

fi

### OUTPUT



**LOGICAL OPERATORS (BOOLEAN OPERATORS)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N** | **OPERATOR** | **OPERATORS IN SHELL** | **DESCRIPTION** |
| 1. | Logical AND | && | Binary operator which returns true if both the operands are true otherwise returns false value |
| 2. | Logical OR | || | Binary operator which returns true if one of the operand or both is true otherwise returns false value |
| 3. | Not Equal to | ! | Unary operator returns true if the operand is false and returns true if the operand is true |

### EXAMPLE OF LOGICAL AND OPERATOR SOURCE CODE

echo " " echo "\tLogical AND Operator"

echo " " echo "Enter the number 1: "

read a

echo "Enter the number 2: " read b

echo "Enter the number 3: " read c

if [ $a -gt $b ] && [ $a -gt $c ] then

echo "$a is bigger than $b and $c" elif [ $b -gt $c ]

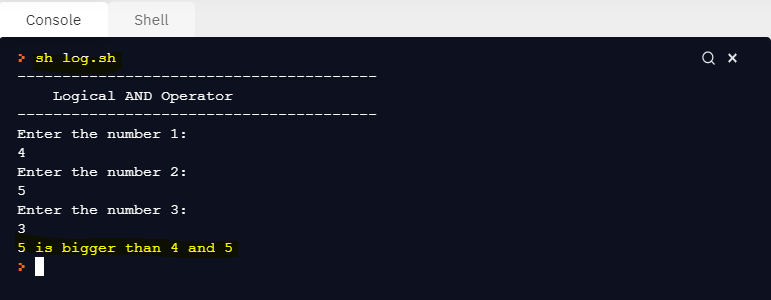
then

echo "$b is bigger than $a and $b" else

echo "$c is bigger than $a and $b"

fi

### OUTPUT



**FILE TYPE OPERATORS**

|  |  |  |
| --- | --- | --- |
| **S.N** | **OPERATOR** | **DESCRIPTION** |
| 1. | -f | Returns true if exists and if it is a regular file (.txt, .c. sh, etc,…) |
| 2. | -d | Returns true if exists and if it is a directory |
| 3. | -e | Returns true if exists |
| 4. | -z | Returns true if file is empty (file has zero length) |
| 5. | -r | Returns true if exists and is readable mode |
| 6. | -w | Returns true if exists and is writable mode |
| 7. | -x | Returns true if exists and is executable mode |

### EXAMPLE OF FILE TEST OPERATORS SOURCE CODE

echo " " echo "\tFile Testing Operator"

echo " " echo "Enter the file name : "

read fp

if [ -e $fp ] then

echo "Object Exists..." if [ -f $fp ]

then

echo "It is a regular file..." echo "Contents:"

echo $(cat $fp) elif [ -d $fp ]

then

echo "It is a directory..." dpath=`pwd`/$fp

echo $dpath

echo "Contents of Directory: " for i in $(ls $dpath)

do

echo $i done

else

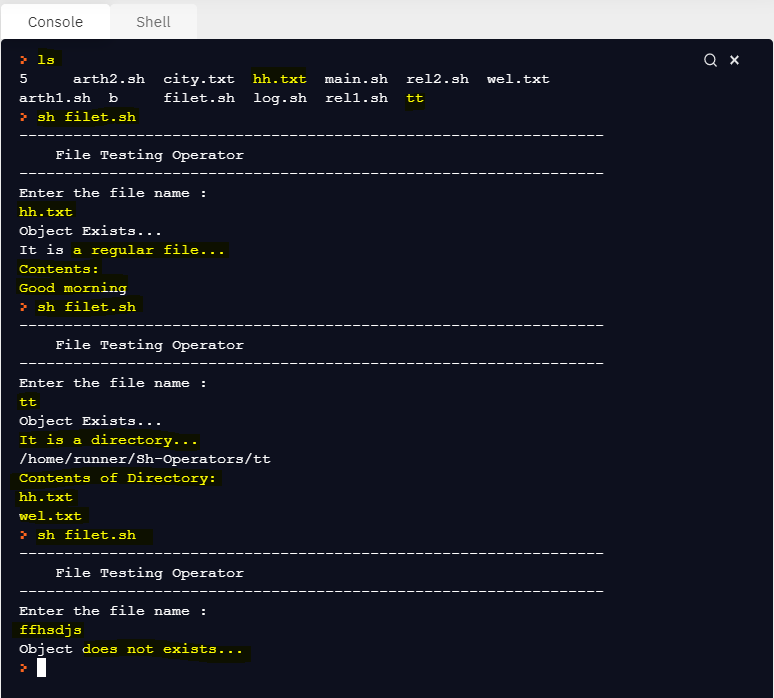
echo "It is a special file..."

fi else

echo "Object does not exists..."

fi

### OUTPUT



**DIRECTORIES**

|  |  |  |
| --- | --- | --- |
| **S.N** | **OPERATOR** | **DESCRIPTION** |
| 1. | ls (OR) ls . | Shows the list of files and folders in current directory |
| 2. | ls .. | Shows the list of files and folders in parent directory |
| 3. | ls / | Shows the list of files and folders in root working directory |
| 4. | ls -l | Shows the files and folders in long listing format |
| 5. | ls -s | Shows the size of files and folders in current directory |

### LISTING FILES AND FOLDERS IN ROOT DIRECTORY SOURCE CODE

(test.sh) echo " " echo "\tFiles and Folders in Root Directory" echo " " for i in $(ls /)

Root Path: $ ls /

do

echo $i done

### OUTPUT



1. **LISTING FILES AND FOLDERS IN PARENT DIRECTORY SOURCE CODE**

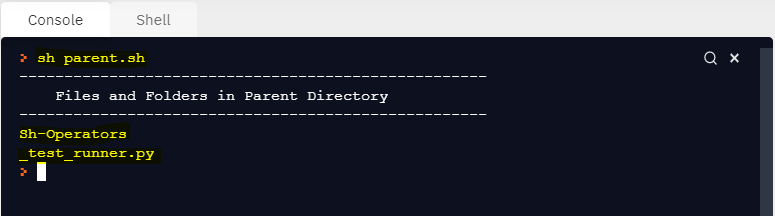
echo " " echo "\tFiles and Folders in Parent Directory" echo " " for i in $(ls ..)

Path of Parent Directory: $ ls ..

do

echo $i done

### OUTPUT



1. **LISTING FILES AND FOLDERS IN CURRENT DIRECTORY SOURCE CODE**

echo " " echo "\tFiles and Folders in Current Directory" echo " " for i in $(ls)

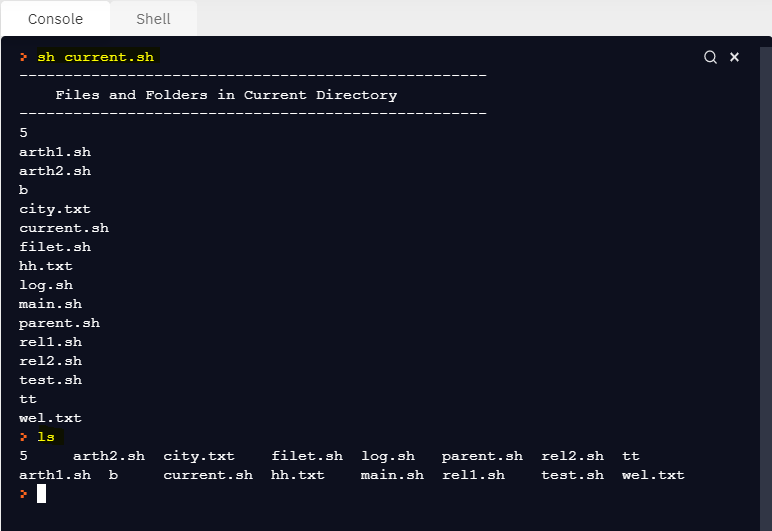


Path of Current Directory: $ ls

do

echo $i done

### OUTPUT

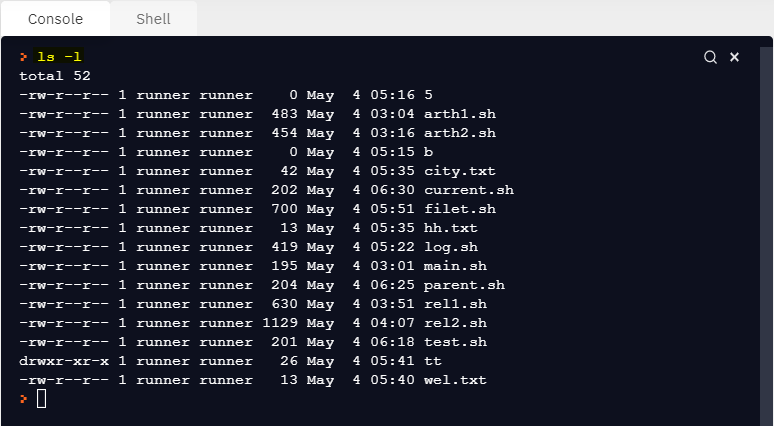


**LIST FILES IN LONG FORMAT**

* The command **ls -l** provides the detailed format of showing the files and folders in current file system.

1. Content Permissions
2. Number of links to the content
3. Owner of the content
4. Group onwer of the content
5. Size of the content (in bytes)
6. Last modified date / time
7. Name of the file / directory name

### Long Listing of Files and Folders



**CONTENT PERMISSION**

* In content permission, the column 1 indicates the file type. They are
  + **-** means for regular file
  + d means for directory
  + b means for special block file
  + c means for special character file
* In content permission, the next column indicates the file permissions such as read, write and execute
* Read  represents the read mode which is code value **4**
* Write  represents the write mode which is code value **2**
* Execute  represents the execute mode which is code value **1**.

### SHELL FUNCTION

* Shell program supports the function which is used to implement the variables as well as execute the linux commands.

### Syntax1

function <name>

{

# user code

}

**Example**

function show

{

# user code

}

### Syntax2

<function-name>()

{

# user code

}

**Example**

show()

{

# user code

}

### CALLING FUNCTION

* Shell function can be called **using its name only.**
* The operation **() should not be used while calling the function**.

### Syntax

$ function-name

**Example**

$ show

### EXAMPLE OF SHELL FUNCTION

**SOURCE CODE**

echo " " echo "\t\tShell Function"

echo " " disp()

{



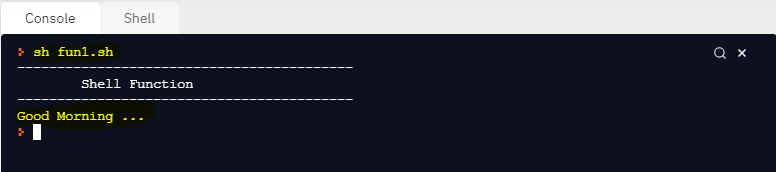
The operator symbol () is not allowed while calling the shell function.

echo "Good Morning ..."

}

# calling function disp

### OUTPUT



**SHELL FUNCTION WITH ARGUMENTS**

* Shell function supports the arguments
* The arguments are given after the function name while calling the function
* Each argument is separated by space
* The positional parameters like $1, $2, etc… will receive the values of function arguments inside the function definition

### EXAMPLE OF SHELL FUNCTION WITH ARGUMENTS SOURCE CODE

echo " " echo "\t\tShell Function with Arguments"

echo " " # creating a shell function

disp()

{

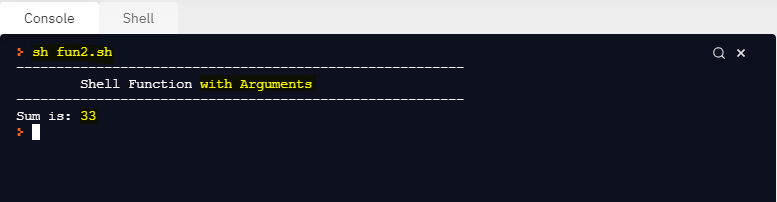
a=$1 b=$2

rs=`expr $a + $b` echo "Sum is: $rs"

}

# calling function with arguments disp 12 21

### OUTPUT



**SHELL FUNCTION WITH RETURN STATEMENTS**

* Like c language, shell function returns the values
* The return keyword is used to return the values in the shell function
* The returned result of calling function can be obtained using the special symbol $? (by default, the return values of the function will be stored to the built-in variable $?)

### NOTE

* It is an important to note that, the shell function will **return a single value**.

### EXAMPLE OF SHELL FUNCTION WITH RETURN VALUE SOURCE CODE

echo " " echo "\t\tShell Function with Return Value"

echo " " a=12

b=23

rs=0

# shell function disp()

{

rs=`expr $a + $b` # return the variable

return $rs

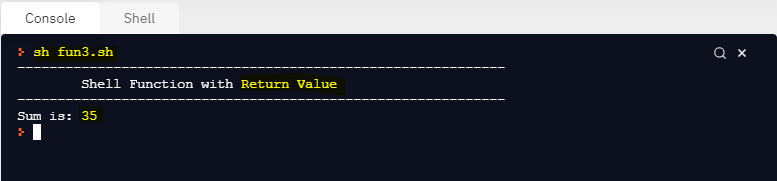
}

# calling function disp

# get the return value from function using $? k=$?

echo "Sum is: $k"

### OUTPUT



**COMMAND SUBSTITUTION**

* In shell, assigning the built-in command to a user defined variable is called as command substitution
* This is done using the special symbol **`command-name`** or **$(command- name)**

### Syntax1

Variable-Name=**`**command-name**`**

**Example**

files=`ls`

# the output of ls command is stored to the

variable called files.

dirpath=”test”

files=`ls $dirpath ` # the contents of test directory are stored

to the variable called files

### Syntax 2

Variable-Name=**$(**command-name**)**

**Example**

files=$(ls)

# the output of ls command is stored to the

variable called files.

### EXAMPLE OF COMMAND SUBSTITUTION SOURCE CODE

echo " " echo "\tCommand Substitution"

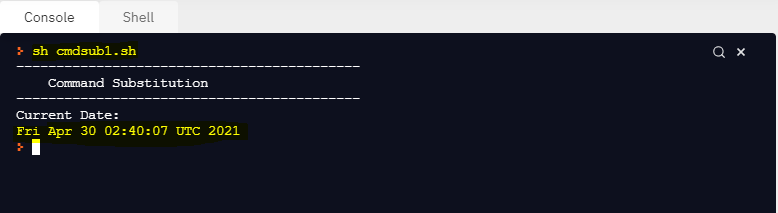
echo " " # date command Substitution

rs=`date`

# k=$(date)

# print the output of date command via variable echo "Current Date: \n"$rs

### OUTPUT



* + 1. **COMMAND SUBSTITUTION-FILE COPYING BETWEEN DIRECTORIES SOURCE CODE**

echo " " echo "\t\tCopying Files B/W Directories"

echo " " # path of source directory path="/home/runner/OS-Lab/d1"



Command Substitution using the symbol ` `

# k=$(ls $path) src=`ls $path`

Path of Source Directory (d1). Parent Path can be get using pwd command.

path of target directory tar="/home/runner/OS-Lab/d5" # for loop

Path of Target Directory (d5).

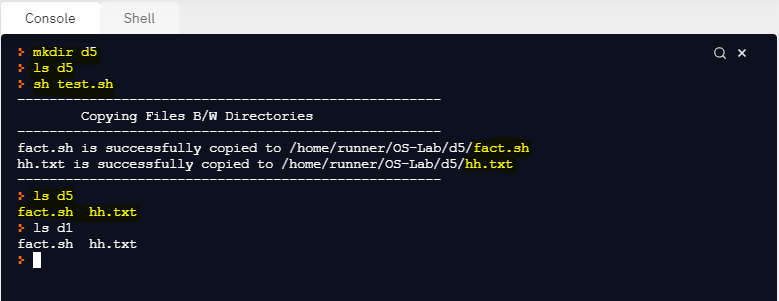
for i in $src do

cp $i $tar

echo "$i is successfully copied to $tar/$i" done

echo " "

### OUTPUT



**RESULT**

Thus the operators and shell functions of the shell programming have been executed successfully.