**SHELL SCRIPTING**

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

# Extended Shell Scripts

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

The shell is, after all, a real programming language, complete with variables, control structures, and so forth.

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.

1. SCRIPT

```
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

SHELL

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 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 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

# Shell 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.

**SAMPLE-2**

```
#!/bin/bash
pwd
ls
```

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 ( _).

The following examples are valid variable names −

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

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

# Defining Variables

variable_name=variable_value

```
NAME="Zara Ali"
VAR1="Zara Ali"
VAR2=100
```

# Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (**$**) −

```
NAME="Zara Ali"
echo $NAME
```

# 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.

```
NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

# 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="Zara Ali"
unset NAME
echo $NAME
```

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.

SPECIAL VARIABLES IN UNIX

we will discuss in detail about special variable in Unix.

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

```
$echo $$
```

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

| Sr.No. | Variable & Description |
|---|---|
| 1 | **$0**<br>The filename of the current script. |
| 2 | **$n**<br>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 $2, and so on). |
| 3 | **$#**<br>**The number of arguments supplied to a script.** |
| 4 | **$***<br>**All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.** |

| | |
|---|---|
| 5 | **$@**<br>**All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.** |
| 6 | **$?**<br>**The exit status of the last command executed.** |
| 7 | **$$**<br>**The process number of the current shell. For shell scripts, this is the process ID under which they are executing.** |
| 8 | **$!**<br>**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 −

```
#!/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 ARG1 ARG2

## Special Parameters $* and $@

There are special parameters that allow accessing all the command-line arguments at once. **$*** and **$@** both will act the same unless they are enclosed in double quotes, **""**.

Both the parameters specify the command-line arguments. However, the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

```
#!/bin/sh

for TOKEN in $*
do
  echo $TOKEN
done
```

```
test.sh Zara Ali 10 Years Old
Zara
Ali
10
Years
Old
```

## Exit Status

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

```
test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
$echo $?
0
$
```

ARRAY

A shell variable is capable enough to hold a single value. These variables are called scalar variables.

Shell supports a different type of variable called an **array variable**. This 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.

# 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 −

```
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 −

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

# 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.

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

The above example will generate the following result −

```
$./test.sh
First Index: Zara
Second Index: Qadir
```

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

```
${array_name[*]}
${array_name[@]}
```

```
NAME[0]="Zara"
NAME[1]="Qadir"
```

```
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

## Operators:

There are various operators supported by each shell.

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

- # Arithmetic Operators
- The following arithmetic operators are supported by Bourne Shell.
- Assume variable **a** holds 10 and variable **b** holds 20 then –
- Show Examples

| 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 |
| = (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 ]**

## 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 –

Show Examples

| 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. |

## Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

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

Show Examples

| 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 –

Show Examples

| 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. |
| **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 –

| | | |
|---|---|---|
| **-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. | |
| **-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. |
| **-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. |

## Shell script – fileop.sh

```
file="/var/www/tutorialspoint/unix/test.sh"

if [ -r $file ]
then
  echo "File has read access"
else
  echo "File does not have read access"
fi

if [ -w $file ]
then
  echo "File has write permission"
else
  echo "File does not have write permission"
fi
if [ -x $file ]
then
  echo "File has execute permission"
else
  echo "File does not have execute permission"
fi

if [ -f $file ]
then
  echo "File is an ordinary file"
else
  echo "This is sepcial file"
fi
if [ -d $file ]
then
  echo "File is a directory"
```

```
else
  echo "This is not a directory"
fi

if [ -s $file ]
then
  echo "File size is not zero"
else
  echo "File size is zero"
fi
```

**CONDITIONS**

While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform the right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here –

- The **if...else** statement
- The **case...esac** statement

# The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if...else** statement –

- if...fi statement

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
Fi

```

## Shell script - if1.sh

```
a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
```

```
then
  echo "a is not equal to b"
fi
```

- if...else...fi statement

```
if [ expression ]
then
  Statement(s) to be executed if expression is true
else
  Statement(s) to be executed if expression is not true
fi
```

```
a=10
b=20

if [ $a == $b ]
then
  echo "a is equal to b"
else
  echo "a is not equal to b"
fi
```

- if...elif...else...fi statement

```
if [ expression 1 ]
then
  Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
  Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
  Statement(s) to be executed if expression 3 is true
else
  Statement(s) to be executed if no expression is true
fi
```

```
a=10
b=20

if [ $a == $b ]
then
  echo "a is equal to b"
elif [ $a -gt $b ]
then
  echo "a is greater than b"
elif [ $a -lt $b ]
```

```
then
  echo "a is less than b"
else
  echo "None of the condition met"
fi
```

Most of the if statements check relations using relational operators discussed in the previous chapter.

Def :

1. Write a shell script to calculate the gross salary of employee.

If basic is less than 3000 than hra-10%, da-25%, and ta – 8%

Else hra- 15, da-27 and ta- 10

Grosssal.sh

2. Write a script to enter the filename and if it is having write permission, append the data in it otherwise display the message.
   Fileperm.sh

## The case...esac Statement

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

There is only one form of **case...esac** statement which has been described in detail here –

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

```
case word in
```

```
  pattern1)
    Statement(s) to be executed if pattern1 matches
    ;;
  pattern2)
    Statement(s) to be executed if pattern2 matches
    ;;
  pattern3)
    Statement(s) to be executed if pattern3 matches
    ;;
  *)
    Default condition to be executed
    ;;
esac
```

```
FRUIT="kiwi"

case "$FRUIT" in
  "apple") echo "Apple pie is quite tasty."
  ;;
  "banana") echo "I like banana nut bread."
  ;;
  "kiwi") echo "New Zealand is famous for kiwi."
  ;;
esac
```

## shell script - swcase.sh

## shell script – swcase1.sh

write a shell script to take the input from the user. If user gives + as an input than addition , subtraction , multiplication and division should be performed.

## Shell script – swcase2.sh

A good use for a case statement is the evaluation of command line arguments as follows –

```
option="${1}"
case ${option} in
  -f) FILE="${2}"
    echo "File name is $FILE"
    ;;
  -d) DIR="${2}"
    echo "Dir name is $DIR"
    ;;
  *)
    echo "`basename ${0}`:usage: [-f file] | [-d directory]"
    exit 1 # Command to come out of the program with status 1
    ;;
esac
```

## LOOP

A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers –

- The while loop

  The **while** loop enables you to execute a set of commands repeatedly until some condition occurs.

```
while command
do
  Statement(s) to be executed if command is true
Done
```

Shell script – wh1.sh

```
a=0

while [ $a -lt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

- The for loop

```
for var in word1 word2 ... wordN
do
  Statement(s) to be executed for every word.
done
```

```
for var in 0 1 2 3 4 5 6 7 8 9
do
  echo $var
done
```

```
for FILE in $HOME/.bash*
do
  echo $FILE
done
```

for1.sh

for2.sh

for3.sh

- The until loop

- The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.
- If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

```
until command
do
  Statement(s) to be executed until command is true
done
```

```
a=0

until [ ! $a -lt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

- The select loop

The **select** loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

## Syntax

```
select var in word1 word2 ... wordN
do
  Statement(s) to be executed for every word.
done
```

```
select DRINK in tea cofee water juice appe all none
do
  case $DRINK in
    tea|cofee|water|all)
      echo "Go to canteen"
      ;;
    juice|appe)
      echo "Available at home"
    ;;
    none)
      break
    ;;
    *) echo "ERROR: Invalid selection"
    ;;
  esac
```

```
done
```

You will use different loops based on the situation. For example, the **while** loop executes the given commands until the given condition remains true; the **until** loop executes until a given condition becomes true.

Once you have good programming practice you will gain the expertise and thereby, start using appropriate loop based on the situation. Here, **while** and **for** loops are available in most of the other programming languages like **C**, **C++** and **PERL**, etc.

## Nesting Loops

All the loops support nesting concept which means you can put one loop inside another similar one or different loops. This nesting can go up to unlimited number of times based on your requirement.

Here is an example of nesting **while** loop. The other loops can be nested based on the programming requirement in a similar way –

## Nesting while Loops

It is possible to use a while loop as part of the body of another while loop.

### Syntax

```
while command1 ; # this is loop1, the outer loop
do
  Statement(s) to be executed if command1 is true

  while command2 ; # this is loop2, the inner loop
  do
    Statement(s) to be executed if command2 is true
  done

  Statement(s) to be executed if command1 is true
done
```

### Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine –

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
  b="$a"
```

```
  while [ "$b" -ge 0 ]  # this is loop2
  do
    echo -n "$b "
    b=`expr $b - 1`
  done
  echo
  a=`expr $a + 1`
done
```

This will produce the following result. It is important to note how **echo -n** works here. Here **-n** option lets echo avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

Nestingloop.sh

```
12345
1234
123
12
1
```

## What is Substitution?

The shell performs substitution when it encounters an expression that contains one or more special characters.

Here, the printing value of the variable is substituted by its value. Same time, **"\n"** is substituted by a new line

```
a=10
echo -e "Value of a is $a \n"
```

```
op woth -e :- Value of a is 10
op without -e :- Value of a is 10\n
```

## Command Substitution

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrates the command substitution

```
DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; uptime`
echo "Uptime is $UP"
```

**SHELL SCRIPT DEFINITIONS –**

SS-1: WRITE A SHELL SCRIPT TO FIND THE MAXIMUM OF 3 NUMBERS.

SS-2: WRITE A SHELL SCRIPT TO FIND THE GRADE OF THE STUDENT IN MARKSHEET EXAMPLE

**SS-3: Accept the string and checks whether the string is palindrome or not.**

```
echo "Enter string"
read str
len=`echo $str |wc -c`
    i=1
    i=$len
    while [ $i -gt 0 ]
    do
       x=`echo $str | cut -c $i`
            temp=$temp$x
     i=`expr $i - 1`
    done
echo $temp
   if [ $str = $temp ]
   then
      echo "pallindrom..!!"
   else
       echo "Not pallindrom..!!"
    fi
```

**SS-4: Write a script to find the sum of the digits of the number**

**SS-5: Accept number and check the number is even or odd**

```
echo " Even no is even or odd"
echo " Enter a number"
read no
```

```
clear
temp=`expr $no % 2`
if [ $temp -eq 0 ]
then
    echo $no "is even"
else
     echo $no "is Odd"
fi
len=`echo $no | wc -c`
len=`expr $len - 1 `
echo "Length of number is " $len
sum=0
rem=0
while [ $no -gt 0 ]
do
  rem=`expr $no % 10`
  sum=`expr $sum + $rem`
  no=`expr $no / 10`
done
echo "Sum of all digits is:" $sum
```

**SS-6: Write a script that accept strings and replace a sub string by another string.**

```
echo -n "Enter string:"
read str
echo "Yor string is :" $str
echo -n "Enter word u want to replace with other word:"
read oldstr
echo -n "Enter new word:"
read newstr
str=`echo $str | sed s/$oldstr/$newstr/g`
echo "After replacing:" $str
```

**SS-7: Write a script that accept the old and new string and replace old string in file.**

**SS – 8 : Accept filename and displays last modification time if file exists, otherwise display appropriate message.**

```
echo "Enter file name"
read fname
```

```
if [ -f $fname ]
then
   echo `ls -l|grep $fname|cut -f10 -d ' '`
else
     echo "File does not exist"
fi
```

**SS-9: Accept filename and displays the permission of that file if file exists otherwise print message**

**SS-10: Fetch the data from a file and display data into another file in reverse order.**

```
echo "enetr file name"
read fname

tac $fname > temp
cat temp
```

**SS-11: Write a script to broadcast a message to a specified user or a group of users logged on any terminal.**

```
echo "Enter user Name"
read uName
user=`who | grep -c $uName`
 if [ $user -gt 0 ]
 then
      echo $user

     write $uName
 else
      echo "\nBrodcast message to all loggin users"
      echo "\nEnter your message"
      echo "\n[Enter ctrl + d to stop]"
 wall
fi
```

**SS-12: Write a script to delete zero sized files from a given directory**

```
echo "DELETE ZERO SIZED FILES"

echo `find . -type f -size 0 -exec

rm {} \; `

ls –l
```

**SS-13: Write a script to display the date, time and a welcome message (like Good Morning should be displayed with 12 hours notation**

```
        date

       hour=`date| cut -c28-29`
echo $hour
time=`date |cut -f6 -d ‘ ‘ `
echo $time
if [ $hour -ge 0 -a $hour -lt 12  -a $time = ‘AM’]
then
       echo "Good Morning..!!"
     elif [ $hour -ge 1 -a $hour -lt 4 -a $time =’PM’ ]
    then
      echo "Good AfterNoon..!!"
   elif [ $hour -ge 4 -a $hour -lt 8 -a $time =’PM’  ]
                  then
          echo "Good Evening..!!"
                   else
            echo "Good Night..!!"
                    fi
```

**SS-14 : Write a script to display the name of all executable files in the given directory.**

```
echo "All Executabel Files from current Directory"
find . -perm /u=r,g=r,o=r
```

**SS-15: Write a script to display the directory in the descending order of the size of each file.**

```
echo "List of all files in descending order from a directory"

echo “Enter Directory “

read direc

cd $direc

ls –Sl
```

**SS-16: Write a script to make following file and directory management operations menu based:**

**Display current directory**

**List directory**

**Make directory**

**Change directory**

**Copy a file**

**Rename a file**

**Delete a file and**

**Edit**

```
echo " 1:Display current Directory"
echo " 2:List Directory"
echo " 3:Make directory "
echo " 4:Change directory "
echo " 5:Copy a file "
echo " 6:Rename a fIle "
echo " 7:Delete a file "
echo " 8:Edit a file "
echo " Enter Your Choice: "
read ch
case $ch in
1)
  pwd ;;

2)
ls  ;;

3) echo "Enter directory name to make "
 read dir_name
 mkdir $dir_name ;;
4)echo " enter directory to change"
read dir1
`cd $dir1`   ;;
```

```
5)
    echo "Enter two file name to copy"
    read f_name1 f_name2
    cp $f_name1 $f_name2
    ;;
6) echo "Enter old file name to rename"

  read fname

  echo "Enter new file name to rename"

  read nfname
  mv $fname $nfname
  ;;
7)
  echo "Enter file name to delete"
  read fname
  rm $fname     ;;
8)echo "Enter file name to edit"
 read fname
cat >> $fname      ;;
 *)echo "You hv entered wrong choice "
                    ;;
           esac
```

**SS-17: Write a script which reads a**

**text file and output the following:**

**Count of character,words and lines.**

**File in reverse. Frequency of particular word in the file.**

**Lower case letter in place of upper case letter.**

```
echo "\n Enter Filename : "
read filen
 echo "----------"  echo "
MENU  " echo "----------
-" echo " a) Count the no. of chars, words, lines."
echo " b) Display a file in reverse."
            echo " c) Count the no. of occurrences of a particular word."
            echo " d) Convert Lower case to UppeCase"
            echo "----------"
            echo "\n Enter Your Choice : "
            read c
            case "$c" in
a) echo "\n Total lines,words,chars "
                        wc $filen
                        ;;
b) tac $filen > temp
                       cat temp ;;


c) echo "\nEnter word to find :"
   read w
     for i in `cat $filen`
   do
```

```
    echo $i >> f1.txt
done
            echo "Total no. of words= \c" ;
                    grep -c $w f1.txt
                    grep $w f1.txt
                    rm f1.txt              ;;

          d)

                  echo "See $filen file for output..."

       cat $filen | tr [a-z] [A-Z]

                          ;;

        *) echo "Invalid choice"



       esac
```