SHELL SCRIPTING

The shell is an interface between the user and the OS itself. The shell also incorporates a powerful programming language that enables the user to exploit the full power and versatility of Unix. It has local and global variables, control instructions, functions,etc. If we are to execute a shell program we don't need a separate compiler. The shell itself interprets the commands in the shell program and executes them. Shell scripts contains a variety of valid Unix commands, however while writing shell programs we will be using these commands in a different way. Instead of using them to perform an isolated task we will combine them effectively to perform complex task. The commands dfspace and shutdown are nothing but shell scripts. Incidentally there are 280 shell scripts that come with the UNIX OS.

Use of Shell Scripts.

To customize your work environment, Automating your daily tasks, Automating repetitive tasks, Executing important system procedures like shutting down the sytem, formatting disk, create a file system on it, mounting the file system., perform the same operation on many files, etc.

Should not use shell scripts when

It is too complex, such as writing an entire billing system, requires a high degree of efficiency, and requires a variety of software tools.

Example of a simple shell script

it has to be stored in a file and then executed on the shell all the three commands will be executed in sequence.

Example:
echo What is your name\?
read name
echo Hello $name. Happy programming.

SHELL VARIABLES

Shell variables are an integral part of shell programming. The variables you use are completely under your control. You can create and destroy any no. of variables as needed.

The **rules for building shell variables** are as follows:

1) A variable name is any combination of alphabets, digits and an underscore(_)
2) No commas or blanks are allowed within a variable name.
3) The first character of a variable name must be either be alphabet or an underscore.
4) Variables names should be of any reasonable length.
5) Variables names are case sensitive. i.e. NAME and name are different.

Shell Keywords

Keywords are the words whose meanings has already been explained to the shell

Keywords cannot be used as variable names. Following is the list of keywords in Borne shell

Name=mandar
Age=20
Dirname=/usr/qq5

Note that there should be no spaces on either side of '='. If space exists, then the shell will try to interpret the value being assigned as a command to be executed. If variable name does not exist, it will be created. It will replace the old value with the new value if the variable already exists.

Variables in UNIX are of two types.
1) User defined variables
2) Unix-defined variables or system variables.
Default primary prompt for PS1 is $
$PS1="What next"
What next

So now every time the system prompts you, it displays not the $ but **What next**

Try this by using echo $shell variable and see the results.

PS2	The system prompt 2, default value is ">"
PATH	Defines the path which the shell must search in order to execute any command or file.
HOME	Stores the default working directory of the user. On entering just a cd, the system understands that HOME is where we want to go and we are back to our default directory. Your result will be "/home/kirtan/"
LOGNAME	Stores the login name of the user. "nihar"
MAIL	Defines the file (along with the path) where the mail of the user is stored. Your result will be /home/nihar/Maildir/

TERM Defines the name of the terminal on which you are working. “ansi”

TZ Defines the name of the time zone in which you are working.

The list of all system variables and their values can be displayed by saying set command at the prompt $set

User defined variables

$a=20
$echo $a
20
$echo a
a

TIPS :

a) All shell variables are string variables. In the statement a=20, the ‘20’ stored in a, is treated not as a number, but as a string of character 2 and 0. Naturally we cannot carry out arithmetic operations on them unless we use a command called expr.

b) A variable may contain more than one word. In such cases, the assignment must be made using double quotes.
$c=“Two words”
$echo $c
Two words

c) We can carry out more than one assignment in a line,
$name=jonny age=10
$echo $name $age

e) A variable which has been defined but has not been given any value is known as a null variable.

```
$d=””
$d=’’
$d=
```

f) On echoing a null variable, only a blank line appears on the screen.

g) If a null variable is used anywhere in a command the shell manages to ignore it. For example

```
$v1=””
$v2=””
$wc –l $v1 $v2 file1
110
```

h) Not only the system variables but also the user-defined variables defined at the $ prompt or in a shell script can be displayed using the set command.

Unchanging variables (Constants)

$a=20
$readonly a
You cannot change the value of a, by making it readonly.

Wiping out Variables

$unset b
with this the variable b and the value assigned to it are simply erased from the shell’s memory. However one should not attempt to do $unset PS1 otherwise the prompt will never come. (BE CAREFULL NOT TO DO THIS)

chmod 744 $1
echo $1 $2 $3 $4 $5 $6 $7 $8 $9
name=$1
set 'who am I'
mv $name $name.$1
$set You have the capacity to learn from mistakes. You will learn a lot in your life.
$echo $*
You have the capacity to learn from mistakes. You will learn a lot in your life.

$* stands for all positional parameters including those beyond 9

Example :
echo $*
shift 1
echo $*
shift 1
echo $*
shift 1
echo $*
shift 1
echo $*

The output will be like
You have the capacity to learn from mistakes. You will learn a lot in your life.

to learn from mistakes. You will learn a lot in your life.

Arithmetic in shell script

```
a=20 b=10
echo `expr $a + $b`
echo `expr $a - $b`
echo `expr $a \* $b`
echo `expr $a / $b`
echo `expr $a % $b`
$a\*\($b + $c\) / $d
```

expr is put within accent graves it is substituted with the output of expr, which is then promptly displayed by the echo statement on the screen.

expr is only capable of carrying out integer arithmetic to perform real number manipulation use the bc command.

ECHO Revised

1)

```
echo Enter the values of a, b and c
echo $a $b $c
```

2)

```
$ echo -e " I like work…\n I can sit and watch it for hours." # use –e option to invoke
I like work…                                                  #back slash character
I can sit and watch it for hours.                             # constants
```

For carriage Return \r, for tab \t, for back space \b, etc.

```
$echo "\07" #beep
$
```

Escape sequences	Meanings
\033[0m	Normal characters
\033[1m	Bold characters
\033[4m	Underlined characters
\033[5m	Blinking characters
\033[7m	Reverse video characters

TPUT command

tput bell
tput clear
echo “Total no. of column on screen = \c”
tput cols
echo “Total no. of rows = \c”
tput lines
echo “This is a normal message”
tput blink
echo “This is a blinking message”
tput bold
echo “This is a bold message”
tput cup 10 20 # positions cursor at 10^{th} row and 20^{th} column.
echo “Testing of tput”
tput smso # starts stand out (bold mode)
echo “the bold has began..”
tput rmso #end stand out mode
tput smul #start mode underline
echo “so also has underline”
tput rmul #end mode underline

REDIRECTION

$cat
$wc

when the commands cat and wc are used without arguments in the above manner it is an indication to the shell that the i/p to each of these commands no longer comes from the file but from a stream. The default source of this stream is keyboard. This stream is known as standard input to the command.
Cat and wc also send the o/p as a character stream. The default destination of this stream is the terminal. This stream is called the standard output of the command. The command is ignorant of the destination of the stream that it has generated. When you enter wrong commands, or when the shell or the command encounters an error, like the non-existence of a file that you are trying to open, certain diagnostic messages are echoed by the system. This constitutes the third stream, and, like the standard output, its default destination is also the terminal. This file is called the standard error.

Standard input

The stream can come from
→ the keyboard, i.e. keyed in by the user. This is the default source anyway.
→ a file (using a feature called redirection)
→ another program (Using the concept of a pipeline)

$

Standard output

$wc infile > newfile
$cat newfile

$cat chap?? > textbook
$who >> newfile # does not disturb existing contents & symbol >> is used to append
$(ls –l; who) > lsfile

when the output of a command is redirected to a file, the output file is created by the shell before the command is executed.

Combining Standard Input and Standard Output

$wc < infile > newfile
$wc>newfile<infile
$>newfile < infile wc

Standard error

Before we proceed further, you should know that each of these three standard files has a number, called a file descriptor, which is used for identification.
Descriptor no 0 is for standard input, 1 for standard output and 2 for standard error.
$cat bar 2>errorfile
$cat errorfile

/dev/null and /dev/tty are the special files in unix.

```
$cal 1998 >/dev/null          # size is always zero
$cat /dev/null
$_
$cat chap100 2>/dev/null          #redirects the standard error
$_
$who >/dev/tty   # the list of current users is sent  to the terminal he
is
                  # currently using.
```

Command substitution

```
$echo The date today is ; date
$ echo The date today is `date`
$echo There are `ls | wc –l` files in the current directory
There are 58 files in the current directory.

$echo "There are `ls | wc –l` files in the current directory"
There are 58 files in the current directory.
```

PIPES and FILTERS

In unix, commands were created to perform single tasks only. If we want to perform multiple tasks in one command, it is not possible. Redirection provides an answer to this problem. But it creates a lot of temporary files which are redundant & occupy disk spaces, pipes and filters are used to overcome this obstacle.

pipes can be thought of as temporary unnamed files, which stores the o/p of one command in memory and passes it as i/p to the next command. The important advantage of pipes are that they prevent us from making temporary files using I/O redirection. Since they are only temporary files, they save disk space unlike the disk files.
$cat text | head -3
The o/p of this command will be the display of the first 3 lines of the file text.

FILTERS.

A filter takes input from the standard Input, processes it and then sends the o/p to the standard o/p. Filters also take input from a file. Filters are used to extract the lines which contain a special pattern, to arrange the contents of a file in a sorted order, to replace the existing characters with some other characters ,etc.
Filters are also used to store the intermediate result of a long pipe. We can extract specific columns of a file and can merge two or more files together using filters.

SORT and GREP commands.

The sort filter arranges the i/p taken from the standard i/p in alphabetical order.
The various options with sort command are as follows.
-r display in reversed alphabetical order
-f usually the digits, alphabets and other special characters taken in as i/p are converted to their ascii value. Then sort arranges them according to their ascii value. The sort command when used

-b to overcome the problem of leading spaces. It is used along with +pos and –pos option + pos –pos to sort the data on specific fields in a given file.

GREP FILTER

This command is used to search for a particular pattern from a file or from the standard i/p and display those lines on the standard o/p.
"Grep" stands for "Global search for REgular Expression"
The various options with the grep
-v displays only those lines which do not match the pattern specified.
-c displays only the count of those lines which match the pattern specified.
-n displays those lines which match the pattern specified along with the line no. at the beginning of the line.
-i displays those lines which matches the pattern specified ignoring the case distinction.

. and * are used in grep aswell. Dot means the same to a regular expression as ? means to the shell.

```
$cat data
        India 6890 Asia
        China       8705 asia
$_
$grep "asia" data
        China       8705 asia
$
```

newa, newb and newc are.
$_
$grep “new[a-c]” new
newa, newb and newc are.
$_
suppose we want to extract those line which end with the character “e” then
$grep “e$” new
…….
To exract those lines which begin with ‘t’ then
$grep “^t” new
…….

EGREP

This is an extension of the grep command. :egrep means extended global search for regular expression.
→ multiple patterns can be searched in a file using a single command line. The multiple patterns should be separated by “|” symbol.

$egrep “india|Nigeria|argentina” data
…
...
$
→ egrep to look out for patterns from a different file
$cat counties
france
India
$egrep –f countries data
france 3333 europe

$_

FGREP

It is fixed grep. This command is used to extract only fixed strings without the use of any regular expression. Fgrep when used with –x option is used to extract those lines which match the string exactly.

PG filter

e.g $ls –l | pg
The output of the ls –l command will be displayed one screen at a time. We can continue viewing by pressing enter.
e.g. $ cat text | pg
The content of the text file will be displayed page by page.

MORE filter

e.g. $ who | more
The function is the same as pg. The difference between the pg and more is that the viewing screenful of the later can be done by pressing space bar while that of the former is done by pressing the enter key.

The CUT command

One particular field from any file or from output of any command can be extracted & displayed using the cut command. One particular character can also be extracted using the –c option of

PASTE command

This command merges the contents of two files into a single file. It reads a line from each file in the list specified & combines them into a single file.

$paste firstname secondname lastname
george Mathew Thomas
Victoria Thomas peter
Slyvia peter marry
$

TEE command

Sometimes in long pipelines, we might get undesired result or error message in some part. We will not be able to identify as to where the error occurred. It is also possible that we may want the o/p of a particular command in a long pipeline to be stored for later use. Such problems are solved by using tee command.

$cat mast | sort | tee temp | cut –d":" –f1
…
…
$cat temp
…
..
$_

TR command

This command is used to translate character taken from the

the bag contains
$_
$cat bag | tr “[a-z]” “[A-Z]”
THE BAG CONTAINS
$_

Using grep to list the directories.
$ls –l | grep “^d”

To display only the first field of the first ten lines of a sorted text file, then
$cat text | sort | head | cut –d “ ” –f1

To extract the file owners & view it page wise
$ls –l | tr –s “ ” | cut –d “ ” –f3 | pg

Shell Programming

shell scripts are also called shell programs, shell procedures, The extension of the shell script is .sh
First you create a shell script in any of the editor let us say vi, then you can execute this shell script in two ways one way is to use the sh command along with the filename, alternatively, you can first use chmod command to make the file executable, and then run it by simply invoking the filename.
$cat script.sh
date
cal

READ : Making Scripts Interactive

Read is the shell's internal tool for taking input from the user,
Example :

```
echo "\nEnter the pattern to be searched: \c"
read pname
echo "\nEnter the file to be used: \c"
read flname
echo "\nSearching for $pname from file $flname\n"
grep "$pname" $flname
echo "\nSelected records shown above\n"
```

The sequence "\c" places the cursor at the end of the echo's argument.
Execute this above shell script and you will be interacted with the options.

REVISED COMMAND LINE ARGUMENTS – POSITIONAL PARAMETERS

This non-interactive method of specifying arguments is quite useful for scripts requiring few inputs. When arguments are specified with a shell procedure, they are assigned to certain special "variables", or rather positional parameters. The 1st argument is read by the shell into the parament $1, the 2nd argument into $2 & so on. We can technically call them shell variables since they are evaluated with a $ before the variable name.

```
grep "$1" $2
echo "\n job over"
```

EXIT STATUS OF A COMMAND

The parameter $? Stores the exit status of the last command. It has the value 0 if the command succeeds, and a non-zero value if it fails.

```
 $grep director emp.lst >/dev/null; echo $?
0
$grep manager emp.lst > /dev/null ; echo $?
1                    #manager does not exist
$grep manager emp3.lst > /dev/null; echo $?
Grep: can't open emp3.lst
2
$_
```

SPECIAL PARAMETERS USED BY SHELL

Shell parameter	significance
$1,$2,…..etc.	The positional parameters.
$*	complete set of positional parameters as a single string.
$#	Number of arguments specified in a command line.
$0	Name of the executed command.
$?	Exit status of last command.
$!	PID of last background job.

LOGICAL OPERATORS

The shell provides two operators to control execution of a command depending on the success or failure of the previous command they are && and ||
With && the second command only executes when the first succeeds. You can use it with the grep command in this way:
$grep 'director' emp1.lst && echo "pattern found in file"

The || operator is used to execute the command following it only when the previous command fails. If you grep a pattern from a file without success, you can notify the failure.
$grep 'manager' emp2.lst || echo "Pattern not found"

EXIT

The exit statement is used to prematurely terminate a program. The execution of the program is halted and control is returned to the shell. You do not need to place this at the end of every shell script because the shell understands when script execution is complete. It is quiet often used with a command when the command fails.

grep "$1" $2 || exit 2 #exit also takes an argument
echo "Pattern found – job over"

argument provided with exit is optional and when you specify one, the script will terminate with a return value of the argument. If no

Simple IF SYNTAX

```
if control command
   Command1
fi
```

Example1:

```
if cp $source $target
then
   echo File copied successfully
fi
```

IF – THEN –ELSE –FI

Example1:

```
if cp $source $target
then
   echo File copied successfully
else
   echo Failed to copy the File
fi
```

Example2:

```
if grep “director” emp.lst
```

```
if test $num –lt 6
then
   echo I used to think I was indecisive, but now I am not so
sure
   echo – anonymous
fi
```

Test command can carry out several types of test

A) numerical test B) string test C) file test

Operator	meanings
-gt	greater than
-lt	less than
-ge	greater than or equal to
-le	less than or equal to
-ne	not equal to
-eq	equal to

```
$x=5; y=7; z=7.2
$test $x –eq $y ; echo $?
1
$test $x –lt $y ; echo $?
0
$test $x –gt $y ; echo $?          # 7.2 is not greater than 7
1
$test $z –eq $y ; echo $?          # 7.2 is equal to 7
0
```

Example :

```
        echo "You have not keyed in 3 arguments"
else
        if grep "$1" $2 > $3
        then
                echo "Pattern found – job over"
        else
                echo "Pattern not found – job over"
        fi
fi
```

SHORT HAND FOR TEST

```
test $x –eq $y
[ $x –eq $y ]
```

Example : In a company an employee is paid as under :
If his salary is less than Rs. 1500, then HRA = 10 % of basic salary and DA= 90% of basic. If his salary is either equal to or above Rs. 1500 then HRA= Rs.500 and DA=98 % of basic salary. If the employee's salary is input through keyboard write a program to find his gross salary.

Solution :

```
echo Enter basic salary
read bs
if [ $bs –lt 1500]
then
        hra='echo $bs \* 10 /100 | bc'
        da='echo$bs\*90/100 | bc'
```

if-elif : Multi way Branching

if-then-elif-then-else-fi, where you can have as many elifs as you want, while the else remains optional.

Example:

```
if [ $# -ne 3 ] ; then          # you can put then in this line too
        echo "You have not keyed in 3 arguments "; exit 3
elif grep "$1" $2 > $3 2>/dev/null ; then
        echo "Pattern found – job over"
else
        echo "Pattern not found – job over"; rm $3
fi
```

STRING TESTS

= means equal to and != means not equal to as in 'c'. like other operators, these also should have white space on either side.

Test	Exit status
-n stg	True if string stg is not null string
-z stg	True if string stg is a null string

```
echo "Enter the string to be searched:\c"
read pname
if [ -z "$pname" ] ; then          # -z checks for a null string
        echo "You have not entered the string" ; exit 1
else
        echo "Enter the file to be used : \c"
        read flname
        if [ ! –n "$flname" ] ; then # ! –n is the same as -z
                echo " You have not entered the filename "; exit 2
        else
                grep "$pname" "$flname" || echo "Pattern not
found"
        fi
fi
```

Test also permits the checking of more than one condition in the same line, using the –a (AND) and –o (OR) operators.

```
if [ -n "$pname" –a –n "$flname" ] ; then
        grep "$panme" "$flname" || echo "pattern not found"
else
        echo "At least one input was a null string"
        exit 1
fi
```

test output is true only if both variables are non-null strings.

FILE TESTS

-f file True if file exists and is a regular file
-r file True if file exists and is readable
-w file True if file exists and is writable
-x file True if file exists and is executable
-d file True if file exists and is a directory
-s file True if file exists and has a size greater than zero

```
$ls –l emp.lst
…..
$ [ -f emp.lst ] ; echo $?
0
$ [ -x emp.lst ] ; echo $?
1
$ [ ! –w emp.lst] || echo “ False that file is not writable”
 False that file is not writable
```

Using the above features, you can design a script that accepts a filename as argument, and then performs a number of tests on it.

```
if [ ! –e $1 ] ; then
        echo “File does not exist”
elif [ ! –r $1 ] ; then
        echo “File Is not readable”
elif [ ! –w $1 ] ; then
        echo “ File is not writable”
else
        echo “File is both readable and writable”
fi
```

```
case expression in
        pattern1)   execute commands  ;;
        pattern2)   execute commands  ;;
        pattern3)   execute commands  ;;
        ……
esac
```

To prepare menu driven shell scripts it is used.

Example :

```
echo –e  “          MENU
              1. List of files\n 2. Processes of user\n3. Today’s  Date
              4. Users of System\n5. Quit to UNIX\n Enter your
              option: \c”
read choice
case “$choice” in
        1) ls –l ;;
        2) ps –f ;;
        3) date ;;
        4) who ;;
        5) exit       # ;; not really required for the last option.
esac
```

IF to CASE

if is given as

```
if [ “$choice” =”y” –o “$choice” = “Y” ]
```

then it can be converted to case structure as

```
echo “Do you wish to continue? (y/n): \c”
read answer
case “$answer” in
```

Case also makes use of Wild card characters as

```
Case "$answer" in
        [yY][eE]*) ;;      #Matches YES, yes, Yes, Yes, yES, etc.
        [nN][oO])  exit ;; #Matches NO, no, nO and No
                *)   echo "Invalid response"#When everything fails
esac
```

* plays here dual role. When used as part of a pattern, it has the usual meaning of the shell wil-card. However, when it is placed as a single character in the last option, it matches any number of characters (or none) not matched by the previous options.

REVISED EXPR

The shell doesn't have any computing features at all; it has to rely on the expr command for that pupose. This makes it relatively slow in operation, but it can be useful in handling simple arithmetic tasks and strings to a limited extent.

```
$x=6 y=2 ; z=`expr $x  + $y`
$echo $z
8
$expr 3 + 5
8
$expr 3 \* 5
15
$x=`expr $x + 1`
$echo $x
```

The message appears exactly 100 seconds after the commands have been invoked. It does not incur significant overheads while it is sleeping.

wait is a shell builtin that checks whether all background processes have been completed. This can be quite useful when you have run a job in the background, and now want to make sure the command is completed so that you can run yet another program. You can use the wait command to wait for the completion of the last background job with or without the process PIDs.

```
wait      #    waits for completion of all background processes
wait 129  #    waits for completion of process PID 129
```


WHILE LOOPING

While, until and for , all of them repeat the instruction set enclosed by certain keywords as often as their control command permits.

SYNTAX

```
ans=y
while [ "$ans" = "y" ]
do
        echo "Enter the code and description : \c"
        read code desc                          #read both together
        echo "$code | $desc" >> newlist  #append a line to file
        echo "Enter any more (y/n)? \c"
        case $anymore in
                Y*|y*) ans =y ;;        # also accepts yes, YES,etc.
                N*|n*) ans=n ;;         # also accepts no, NO, etc.
                    *) ans=y ;;         #any other reply means y
        esac
done
```

Setting up an infinite loop

```
while true ; do         #this form is also permitted
        df –t       #df reports the free space on the disk
        sleep 300
done &                  # & after done runs loop in background
```

Once you have started this program in the background, you can continue your other work, except that every fine minutes you will find your screen filled with df output. To kill this process, you have to use kill $!, which kills the last background job.

UNTIL

Until statement complements the while construct in the sense that

until [-r invoice.lst] #until the file invoice.lst can be read

FOR

No next statement is used here neither the step size can be specified. Unlike while and until, it does not test a condition, but uses a list instead.

The SYNTAX of for is as follows.

```
for variable in list
do
        Execute commands            #Loop body
done
```

The loop body is executed as many times as there are items in the list.

Example

```
$for x in 1 2 4 5 #list has 4 strings
>do
>       echo"The value of x is $x"
>done
The value of x is 1
The value of x is 2
The value of x is 4
The value of x is 5
```

THE LIST

```
$for file in *.c ; do
> cc –o $file{x} $file
> done
```

list from positional parameters

```
$cat emp6.sh
for pattern in $*
do
        grep “$pattern” emp.lst || echo “Pattern $pattern not found”
done
```

list from command substitution

for file in `cat clist` #this method is most suitable when list is large

MORE FILE ATTRIBUTES

To See the inode no.

$ ls –il filename #gives you the inode no. of the file and directories in long listing form

To change the owner

```
$ls –l note
…..
$chown patel note ; ls –l note
….
```

Listing by modification and access times

ls –lt # The time of last modification
ls –lut # The time of last access

To change the time stamp

TOUCH command is widely used to create lot of empty files if needed to be created.
Also it is used to change the timing of the creation of files.

$touch 10 emp.lst ; ls –l emp.lst
$ls –lu emp.lst
$touch –m 4 emp.lst ; ls –l emp.lst
$touch –a 4 emp.lst ; ls –lu emp.lst
-a option changes the access time

links

files are linked with ln command, which takes 2 filenames as arguments.
$ln emp.lst employee
the –i option of ls shows that they have the same inode no, meaning that they are actually one and the same file.
$ ls –li emp.lst employee
..

Read permission

$ ls –ld progs

$chmod 555 progs ; ls –ld progs
….
$cp emp.lst progs
…

Execute Permission

$chmod 666 progs ; ls –ld progs
…
$cd progs

The device

$ ls –l /dev
…..
The device files are grouped into mainly two categories character device and block device. Identified by b or c in the first column.

FEW ILLUSTRATIVE SHELL SCRIPTS THAT WILL MAKE YOU CONFIDENT WITH SHELL PROGRAMMING ARE GIVEN BELOW. EXECUTE IT AND SEE THE EFFECTS.

Example 1:
To accept a filename as input and if it a directory display its contents and revoke the execute permission for group and others.

```
if [ -d $1 ]
then
        echo "$1 is a directory"
```

```
        echo "displaying the changed permissions for files starting
with b"
        ls –l b*
elif [ -f $1 –a –r $1 ]
then
        echo "$1 is an ordinary file displaying its contents..."
        cat $1 | pg
else
        echo "$1 does not exist"
fi
```

Example 2:
To display file permissions along with their names

```
var=$1
for k in $*
do
        ls –l $1 | tr –s " " | cut –d " " –f1,9
        shift
done
```

Example 3:
To accept no. less than 50 & to display their squares.

```
ans=" "
echo "Enter value for ans (y/Y)"
read ans
while [ $ans = y –o $ans = Y ]
do
```

```
        else
                echo “No not in the range”
        fi
        wish = “  “
        echo “Do you wish to continue ? (y/n)”
        read wish
        if [ $wish = y –o $wish = Y ]
        then
                continue
        else
                echo “Thank you for reading my notes and playing with
shell..”
                break
        fi
done
```

Example 4:
Script that display name of those files which have multiple links.

```
if [ $# -eq 0 ]
then
        find –links 2 –print
elif [ $# -eq 1 ]
then
        i=2
        while [ $i –ne $1 ]
        do
                find –links $i –print
```

Example 5:
Script that can be used in place of cd command and when used, it gives the effect of pg of DOS i.e. the current directory becomes the prompt (hence no need to give pwd command)

```
echo “Enter the name of the Directory to be changed”
read dir
if [ -d $dir ]
then
        cd $dir
        echo “The current Directory is “
        pwd; ps1=’[$pwd]`
else
        echo “Directory not found”
fi
```

Example 6:
Script which reads a text file and outputs the following
a) count of characters, words, and lines.
b) Frequency of particular word in the file
c) Lower case letters in place of upper case alphabets.

```
while true
do
echo          -: FACILITIES OF UNIX :-
echo 1. count no. of char, words, lines
echo 2. frequency of particular words in file
echo 3. lower case into upper case
echo 4. exit
echo enter your choice :
```

```
        wc –l –w –c $filename
        else
        echo not possible
        fi ;;
2)      echo Enter the filename from which you want to grep
        read filename
        echo Enter the word you want to grep
        read grpname
        if [ -f $filename ]
        then
                grep –c $grpname $filename
                echo grep success
        else
                echo grep failure
        fi ;;
3)      echo Enter the file name which you want to translate
        read trifle
        if [ -f $trfile ]
        then
                tr “[a-z]” “[A-Z]” <$trfile | cat >$trfile
        else
                echo not possible
        fi ;;
4)      exit ;;
*)      echo wrong choice , try again ;;
esac
done
```

```
echo “Enter name of First Directory : \c”
read dir1
echo “Enter name of Second Directory : \c”
read dir2
echo “Enter name of file : \c”
read file1
path1=`find $dir1 –level 4 –name “$file1”`
path2=`find $dir2 –level 4 –name “$file1”`
echo $path1
echo $path2
ans=`cmp –s $path1 $path2 ; echo $?`
echo “Value of comparison is “ $ans
if test $ans –eq 0
then
        cat $path1 > file3
else
        echo “File names are not same”
fi
```

Example 8:
To find the no. of ordinary files and directories.

```
if [ -f $1 ]
then
        echo “$1 is an ordinary file”
        exit
fi
if [ -d $1 ]
then
        echo “$1 is a directory”
```

Example 9:
If cost price and selling price of an item is input through keyboard, write a program to determine whether the seller has made profit or loss. Also determine how much profit or loss incurred.

```
# To find profit or loss

echo -e "Enter the cost price : \c"
read cost
echo -e "Enter the selling price : \c"
read sel

pl=`expr $sel - $cost`

if [ $pl -lt 0 ]
then
    echo "Seller had loss"
    echo "Loss of Rs. $pl"
else
    echo "Seller had profit"
    echo "Profit of Rs. $pl"
fi
```

:OUTPUT:

```
[SAMIR@www mca25]$ sh ex1.sh
Enter the cost price : 1000
Enter the selling price : 1100
Seller had profit
Profit of Rs. 100
```

4. Exit

 # Menu driven program using case structure

```
echo "1.  Contents of current directory"
echo "2.  List of users who currently logged in"
echo "3.  Present working directory"
echo "4.  Exit"
echo -e " Enter your choice  : \c "
read ch

case $ch in

  1) ls ;;
  2) who ;;
  3) pwd ;;
  4) exit ;;
  *) echo "wrong choice"
esac
```

:OUTPUT:

1. Contents of current directory
2. List of users who currently logged in
3. Present working directory
4. Exit

Enter your choice : 1

copy.sh greet math.sh p1.sh p6.sh pat3.sh pat8.sh timing

Example 11:

Pattern: For n=4

```
   1
  2 3
 4 5 6
7 8 9 10
```

```
# Shell script to print triangle

echo -e "Enter the number : \c"
read n
l=1
for((i=1;i<=n;i++))
 do
  for((k=n-1;k>=i;k--))
   do
     echo -e " \c"
   done
  for((j=1;j<=i;j++))
   do
     echo -e " $l\c"
     l=`expr $l + 1`
   done
  echo " "
 done
```

:OUTPUT:

```
Enter the number : 4
          1
        2   3
      4   5   6
      7   8   9  10
```

Example 12:

Check first argument is string or not.

```
# to check whether argument is string or not

if [ $# -ne 1 ]
then
   echo " Wrong number of arguments"
   exit 1
fi
len=`expr $1 : ".*"`
for((i=0;i<len;i++))
do
  ch=`expr "$1" : ".\{$i\}\(.\)"`
done

case $ch in
  [a-z]) echo 'It is string';;
  *)  echo 'Not string'
esac
```

ADVANCED SHELL SCRIPTING

```
sh join.sh          # in linux
bash join.sh              # This is also correct
sh < join.sh
```

You cannot use sh < a.out.

EXPORT: EXPORTING SHELL VARIABLES

When a process is created by the shell, it makes available certain features of its won environment to the child. The created process (i.e. the command) can also make use of these inherited parameters for it to operate. These parameters include
Pid of the parent process
The user and group owner of the process
The current working directory
The three standard files
Other open files used by the parent process
Some environment variables available in the parent process.

By default, the values stored in the shell variables are local to the shell, i.e. they are available only in the shell in which they are defined. They are not passed to a child shell. But the shell can also export these variables recursively to all child processes so that, once defined, they are available globally. This is done with the export command.

```
$ cat var.sh
echo The value of x is $x
```

The new value of x is 20
$echo $x #value set inside the script doesn't affect value outside script
10

$x=10 ; export x
$var.sh
The value of x is 10 #value outside script now visible here
The new value of x is 20
Command grouping and the cd Command
$pwd
/usr/samir
$ (cd progs ; pwd)
/usr/samir/progs
$pwd
/usr/samir #back to original directory

$pwd
/usr/samir
${ cd progs ; pwd
> }
/usr/samir/progs
$pwd
/usr/samir/progs # directory change is now permanent

TIP : The closing brace must be on a separate line by itself. If, however, you want both the braces in the same line, simply

The ( ) operators execute the enclosed commands in a sub-shell, while the { } operators do the same in the current shell.

EXPR revisited

Determining the length of the string.

```
$expr "abcdefghijkl" : '.*'
12
```

space on either side of : required (.*) is used to count the occurrences of any character

This feature is extremely useful in validating data entry.

Consider that you want to validate the name of a person accepted through the keyboard so that it doesn't exceed, say 20 characters in length. The following expr sequence can be quite useful for this task.

```
while echo "Enter your name : \c"; do
        read name
        if [ `expr "#name" : '.*'` -gt 20 ] ; then
                echo "Name too long"
        else
                break
        fi
done
```