“Shell Programming Lab”

LAB MANUAL

Prepared by

Hirendra Deora

Assistant Professor

DEPARTMENT OF COMPUTER ENGINEERING & IT

VYAS COLLEGE OF ENGINEERING & TECHNOLOGY

JODHPUR (RAJASTHAN)

March 2013

SYLLABUS

6CS8 SHELL PROGRAMMING LAB (Computer Engg)
Max. Marks=50

1. Use of Basic UNIX Shell Commands: ls, mkdir, rmdir, cd, cat, touch, file, wc, sort, cut, grep, dd, dfspace, du, ulimit
2. Commands related to inode, I/O redirection and piping, process control commands, mails.
3. Shell Programming: Shell script exercises based on following:
   (i) Interactive shell scripts (ii) Positional parameters (iii) Arithmetic (iv) if-then-fi, if-then- else-fi, nested if-else (v) Logical operators (vi) else + if equals elif, case structure (vii) while, until, for loops, use of break
4. Write a shell script to create a file. Follow the instructions
   (i) Input a page profile to yourself, copy it into other existing file;
   (ii) Start printing file at certain line
   (iii) Print all the difference between two file, copy the two files.
   (iv) Print lines matching certain word pattern.
5. Write shell script for-
   (i) Showing the count of users logged in,
   (ii) Printing Column list of files in your home directory
   (iii) Listing your job with below normal priority
   (IV) Continue running your job after logging out.
6. Write a shell script to change data format. Show the time taken in execution of this script.
7. Write a shell script to print files names in a directory showing date of creation & serial number of the file.
8. Write a shell script to count lines, words and characters in its input (do not use wc).
9. Write a shell script to print end of a Glossary file in reverse order using Array. (Use awk tail)
10. Write a shell script to check whether Ram logged in, Continue checking further after every 30 seconds till success.
11. Write a shell script to compute gcd lcm & of two numbers. Use the basic function to find gcd & LCM of N numbers.
12. Write a shell script to find whether a given number is prime. Take a large number such as 15 digits or higher and use a proper algorithm.

SYLLABUS

6EC8 SHELL PROGRAMMING LAB (Electronics & Communication Engineering)
Max. Marks=75

1. Use of Basic UNIX Shell Commands: ls, mkdir, rmdir, cd, cat, touch, file, wc, sort, cut, grep, dd, dfspace, du, ulimit
2. Commands related to inode, I/O redirection and piping, process control commands, mails.
3. Shell Programming: Shell script exercises based on following:
   (ii) Interactive shell scripts (ii) Positional parameters (iii) Arithmetic (iv) if-then-fi, if-then- else-fi, nested if-else (v) Logical operators (vi) else + if equals elif, case structure (vii) while, until, for loops, use of break
4. Write a shell script to create a file. Follow the instructions
   (i) Input a page profile to yourself, copy it into other existing file;
   (ii) Start printing file at certain line
   (iii) Print all the difference between two file, copy the two files.
   (iv) Print lines matching certain word pattern.
5. Write shell script for-
   (i) Showing the count of users logged in,
   (ii) Printing Column list of files in your home directory
   (iii) Listing your job with below normal priority
   (IV) Continue running your job after logging out.
6. Write a shell script to change data format. Show the time taken in execution of this script.
7. Write a shell script to print files names in a directory showing date of creation & serial number of the file.
8. Write a shell script to count lines, words and characters in its input (do not use wc).
9. Write a shell script to print end of a Glossary file in reverse order using Array. (Use awk tail)
10. Write a shell script to check whether Ram logged in, Continue checking further after every 30 seconds till success.
11. Write a shell script to compute gcd lcm & of two numbers. Use the basic function to find gcd & LCM of N numbers.
12. Write a shell script to find whether a given number is prime. Take a large number such as 15 digits or higher and use a proper algorithm.

Chapter 1

Introduction & Overview

1.1 Operating System

An **operating system** (**OS**) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system. Application programs usually require an operating system to function.

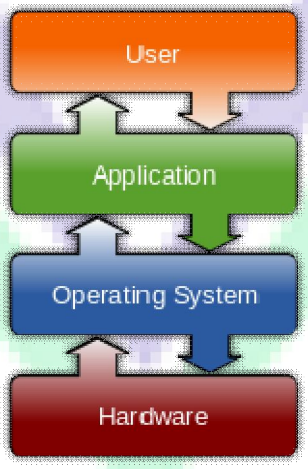


Fig 1.1 Operating System

1.2 Types Of Operating System

a. **Real-time**
A real-time operating system is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior. The main objective of real-time operating systems is their quick and predictable response to events. They have an event-driven or time-sharing design and often aspects of both. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts.

b. **Multi-user**
A multi-user operating system allows multiple users to access a computer system at the same time. Time-sharing systems and Internet servers can be

classified as multi-user systems as they enable multiple-user access to a computer through the sharing of time. Single-user operating systems have only one user but may allow multiple programs to run at the same time.

c. **Multi-tasking vs. single-tasking**
A multi-tasking operating system allows more than one program to be running at a time, from the point of view of human time scales. A single-tasking system has only one running program. Multi-tasking can be of two types: pre-emptive and co-operative. In pre-emptive multitasking, the operating system slices the CPU time and dedicates one slot to each of the programs. Unix-like operating systems such as Solaris and Linux support pre-emptive multitasking, as does AmigaOS. Cooperative multitasking is achieved by relying on each process to give time to the other processes in a defined manner. 16-bit versions of Microsoft Windows used cooperative multi-tasking. 32-bit versions of both Windows NT and Win9x, used pre-emptive multi-tasking. Mac OS prior to OS X used to support cooperative multitasking.

d. Distributed
distributed operating system manages a group of independent computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system.

e. Embedded
Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

f. Time-Sharing
Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting for cost allocation of processor time, mass storage, printing, and other resources.

1.3 Examples of Operating System

a. UNIX and UNIX-like operating systems

Unix was originally written in assembly language. Ken Thompson wrote B, mainly based on BCPL, based on his experience in the MULTICS project. B was replaced by C, and Unix, rewritten in C, developed into a large, complex family of inter-related operating systems which have been influential in every modern operating system.

The *UNIX-like* family is a diverse group of operating systems, with several major sub-categories including System V, BSD, and Linux. The name "UNIX" is a trademark of The Open Group which licenses it for use with any operating system that has been shown to conform to their definitions. "UNIX-like" is commonly used to refer to the large set of operating systems which resemble the original UNIX.
Unix-like systems run on a wide variety of computer architectures. They are used heavily for servers in business, as well as workstations in academic and engineering environments. Free UNIX variants, such as Linux and BSD, are popular in these areas.

Four operating systems are certified by the The Open Group (holder of the Unix trademark) as Unix. HP's HP-UX and IBM's AIX are both descendants of the original System V Unix and are designed to run only on their respective vendor's hardware. In contrast, Sun Microsystems's Solaris Operating System can run on multiple types of hardware, including x86 and Sparc servers, and PCs. Apple's OS X, a replacement for Apple's earlier (non-Unix) Mac OS, is a hybrid kernel-based BSD variant derived from NeXTSTEP, Mach, and FreeBSD.

Unix interoperability was sought by establishing the POSIX standard. The POSIX standard can be applied to any operating system, although it was originally created for various Unix variants.

b. Linux and GNU

Linux (or GNU/Linux) is a Unix-like operating system that was developed without any actual Unix code, unlike BSD and its variants. Linux can be used on a wide range of devices from supercomputers to wristwatches. The Linux kernel is released under an open source license, so anyone can read and modify its code. It has been modified to run on a large variety of electronics. Although estimates suggest that Linux is used on 1.82% of all personal computers, it has been widely adopted for use in servers and embedded systems (such as cell phones). Linux has superseded Unix in most places, and is used on the 10 most powerful supercomputers in the world. The Linux kernel is used in some popular distributions, such as Red Hat, Debian, Ubuntu, Linux Mint and Google's Android.

The GNU project is a mass collaboration of programmers who seek to create a completely free and open operating system that was similar to Unix but with completely original code. It was started in 1983 by Richard Stallman, and is responsible for many of the parts of most Linux variants. Thousands of pieces of software for virtually every operating system are licensed under the GNU General Public License. Meanwhile, the Linux kernel began as a side project of Linus Torvalds, a university student from Finland. In 1991, Torvalds began work on it, and posted information about his project on a newsgroup for computer students and programmers. He received a wave of support and volunteers who ended up creating a full-fledged kernel. Programmers from GNU took notice, and members

of both projects worked to integrate the finished GNU parts with the Linux kernel in order to create a full-fledged operating system.

c. Microsoft Windows

Microsoft Windows is a family of proprietary operating systems designed by Microsoft Corporation and primarily targeted to Intel architecture based computers, with an estimated 88.9 percent total usage share on Web connected computers. The newest version is Windows 8 for workstations and Windows Server 2012 for servers. Windows 7 recently overtook Windows XP as most used OS.

Microsoft Windows originated in 1985 as an operating environment running on top of MS-DOS, which was the standard operating system shipped on most Intel architecture personal computers at the time. In 1995, Windows 95 was released which only used MS-DOS as a bootstrap. For backwards compatibility, Win9x could run real-mode MS-DOS and 16 bit Windows 3.x drivers. Windows ME, released in 2000, was the last version in the Win9x family. Later versions have all been based on the Windows NT kernel. Current versions of Windows run on IA-32 and x86-64 microprocessors, although Windows 8 will support ARM architecture. In the past, Windows NT supported non-Intel architectures.

Server editions of Windows are widely used. In recent years, Microsoft has expended significant capital in an effort to promote the use of Windows as a server operating system. However, Windows' usage on servers is not as widespread as on personal computers, as Windows competes against Linux and BSD for server market share

1.4 Filesystem architecture of Linux

Directory	Content
/bin	Common programs, shared by the system, the system administrator and the users.
/boot	The startup files and the kernel, vmlinuz. In some recent distributions also grub data. Grub is the GRand Unified Boot loader and is an attempt to get rid of the many different boot-loaders we know today.
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc	Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/initrd	(on some distributions) Information for booting. Do not remove!
/lib	Library files, includes files for all kinds of programs needed by the system and the users.
/lost+found	Every partition has a lost+found in its upper directory. Files that were saved

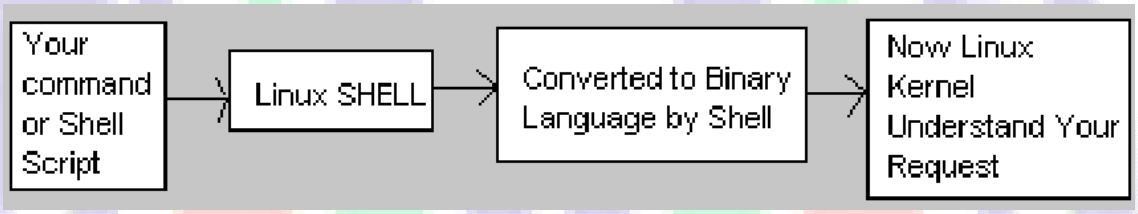
Directory	Content
	during failures are here.
/misc	For miscellaneous purposes.
/mnt	Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
/net	Standard mount point for entire remote file systems
/opt	Typically contains extra and third party software.
/proc	A virtual file system containing information about system resources. More information about the meaning of the files in proc is obtained by entering the command **man *proc*** in a terminal window. The file proc.txt discusses the virtual file system in detail.
/root	The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the *root* user.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

Chapter 2

Basic Linux Commands

2.1 Shell

Computer understand the language of 0's and 1's called binary language, In earlydays of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in O/s there is special program called Shell. Shell accepts your instruction or commands in English and translates it into computers native binary language.



2.1.1 Type of Shell

BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	--
TCSH	Type $ man tcsh	--	TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH).

2.2 Linux Commands

2.1.1 The Manual (terminal mode)

man

This command brings up the online UNIX manual. Use it on each of the commands below.

Usage: man [command name]

eg: man pwd You will see the manual for the pwd command.

2.1.2 File Handling Commands

mkdir – make directories

create directory[ies] if they are not already exists.

Usage: mkdir <DIREECTORY NAME>

eg. mkdir hiren

ls – list directory contents

Usage: ls [OPTION]... [FILE]...

eg. ls Listing Directory contents including only file names

ls -l Long listing of directory contents including user permissions, number of links, size of file or directory, day and time of last modification and file or directory name.

ls –a Listing all files including hidden files

cd – changes directories

Usage: cd [DIRECTORY NAME]

Eg. cd hiren

pwd – print working directory

Shows what directory (folder) you are in.

In Linux, your home directory is /home/username.

Usage: pwd

eg. pwd show present working directory

/home/username

rmdir- Remove directories

remove directory[ies] if they are empty.

Usage: rmdir [DIRECTORY NAME]

Eg. rmdir hiren remove hiren directory if this is empty.

rm-remove files or directories

remove file[s] or directory[ies]

Usage:rm –[option] [DIRECTORY NAME OR FILE NAME]
Eg.
rm –i [FILENAME] remove file interactively. This will ask before removing file.
rm –f [FILENAME] removefile forcefully.
rm –r [FILENAME] recusively remove non empty directory.

mv-move

move or rename files or directories
Usage: mv [SOURCE DIRECTORY] [DESTINATION DIRECTORY]
mv [OLD FILENAME] [NEW FILENAME]
Eg. mv linixdir hiren renaming or moving directory linixdir as hiren. After execution of command, the destination files are only available.

cp – copy

copy files and directories
Usage: cp [OPTION] SOURCE FILE] [DESTINATION FILE]
eg. cp sample.txt sample_copy.txt After execution of command, the both source and destination files are available.

2.2.3 Text Processing

cat – concatenate files and print on the standard output

Usage: cat [OPTION] [FILE]...
eg. cat file1.txt file2.txt
cat when supplied with more than one file will concatenate the files without any header information.
.

cat - used to display the contents of a small file on terminal

usage: cat [file name]

cat- To create file

Usage: cat > [file name]
NOTE: Press and hold CTRL key and press D to stop or to end file (CTRL+D)

cat – to apend text at the end of file

Usage: cat >> [file name]
NOTE: Press and hold CTRL key and press D to stop or to end file (CTRL+D)

echo – display a line of text

Usage: echo [OPTION] [string] ...
eg. echo I love India
echo $HOME

wc -
ommand is used to count lines, words and characters, depending on the option used.
Usage: wc [options] [file name]

You can just print number of lines, number of words or number of charcters by using followi ng
options:
-l : Number of lines
-w : Number of words
-c : Number of characters
Eg. wc file.txt count number of lines, words and character (including whitespaces , newline etc) in a file.

grep - print lines matching a pattern
Usage: grep [OPTION] PATTERN [FILE]...
eg. grep -i apple sample.txt
Options:
-i case-insensitive search
-n show the line# along with the matched line
-v invert match, e.g. find all lines that do NOT match
-w match entire words, rather than substrings

2.2.4 Process Management

kill –
kill ends one or more process IDs. In order to do this you must own the process or be designated a privileged user. To find the process ID of a certain job use ps.
Usage: kill [options] IDs
Eg. kill 1234

ps-
The "ps" command (process statistics) lets you check the status of processes that are running on your Unix system.
Usage:ps The ps command by itself shows minimal information about the processes you are running. Without any arguments, this command will not show information about other processes running on the system.

2.3 Introduction to Vi

Under Linux, there is a free version of Vi called Vim (Vi Improved). Vi (pronounced vee- eye) is an editor that is fully in text mode, which means that all actions are carried out with the help of text commands. This editor, although it may appear of little practical use at first, is very powerful and can be very helpful in case the graphical interface malfunctions.

Syntax : vi name_of_the_file

Once the file is open, you can move around by using cursors. Press I to switch to insert Mode. Press escape key to switch to command mode,

Basic commands

Description

:q Quit the editor (without saving)

:q! Forces the editor to quit without saving

: wq Saves the document and quits the editor

: filename Saves the document under the specified name

: set nu set serial number to lines

2.4 How to write shell script

Following steps are required to write shell script:

(1) Use any editor like vi or mcedit to write shell script.

(2) After writing shell script set execute permission for your script as follows

syntax:

chmod permission your-script-name

Examples:

$ chmod +x your-script-name

$ chmod 755 your-script-name

Note: This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).

ls -l command earlier presented a long of listing file with a line like the following for each file:

-rw-r--r-- 1 root user 0 2009-04-28 08:26 newfile.txt

Here the first character in the first column (-) indicates that the file is a normal file. The next 9 characters indicate the access permissions for the file. The next set of 9 characters is divided into3 groups of 3 cha- racters. Purpose of these characters is as under:

(-) represents no permission

(r) represents 'read' permission

(w) represents 'write' permission
(x) represents 'execute' permission

Permission	Octal number	Equivalent Symbol
Read	4	r--
Write	2	-w-
Execute	1	--x

The three group represents user (owner of the file), group(to which the owner belongs) and others (any other user of the system) respectively. Three characters in each group are for 'read', 'write' and 'execute' permission respectively.

In our example, the owner has 'read' and 'write' permission for the file and everyone else has only read permission. For a normal file, read, write and execute permissions are obvious. For a directory, read and write permissions mean that to read the contents of the directory and create new entries in the directory. Execute permission means that one can search in the directory but not read from or write to the directory.

You can use the chmod command to change the access permissions of a file or a directory. To specify permissions for a file with chmod, any of the following two methods can be used.

Symbol	Meaning
u	User
g	Group
o	Other
a	All (equals to ugo)
+	Add Permission
-	Remove a permission
r	Read Permission
w	Write permission
x	Execute permission

Syntax: chmod u+x filename

(3) Execute your script as
syntax:
bash your-script-name
sh your-script-name
./your-script-name

Examples:
$ bash bar
$ sh bar
$ ./bar

Exercise:

1. Verify that you are in your home directory.
2. Make the directory adir using the following command:

mkdir adir

3. List the files in the current directory to verify that the directory adir has been made correctly.
4. Change directories to adir.
5. Verify that you have succeeded in moving to the adir directory.
6. Verify that the file testfile exists.
7. List the contents of the file testfile to the screen.
8. Make a copy of the file testfile under the name secondfile.
9. Verify that the files testfile and secondfile both exist.
10. List the contents of both testfile and secondfile to the monitor screen.
11. Delete the file testfile.
12. Verify that testfile has been deleted.
13. Clear the window.
14. Rename secondfile to thefile.
15. Issue the command to find out how large thefile is. How big is it?
16. Copy thefile to your home directory.
18. Remove thefile from the current directory.
19. Verify that thefile has been removed.
20. Copy thefile from your home directory to the current directory.
21. Verify that the file has been copied from your home directory to the current directory.

Chapter 3

Shell Scripts-I

3.1 Variables

3.1.1 Variables

When a script starts all environment variables are turned into shell variables. New variables can be instantiated like this:

```
name=value
```

You must do it exactly like that, with no spaces either side of the equals sign, the name must only be made up of alphabetic characters, numeric characters and underscores; it cannot begin with a numeric character.

3.1.2 Command Line Arguments

Command line arguments are treated as special variables within the script, the reason I am calling them variables is because they can be changed with the **shift** command. The command line arguments are enumerated in the following manner *$0*, *$1*, *$2*, *$3*, *$4*, *$5*, *$6*, *$7*, *$8* and *$9*. *$0* is special in that it corresponds to the name of the script itself. *$1* is the first argument; *$2* is the second argument and so on. To reference after the ninth argument you must enclose the number in brackets like this *${nn}*. You can use the **shift** command to shift the arguments 1 variable to the left so that *$2* becomes *$1*, *$1* becomes *$0* and so on, *$0* gets scrapped because it has nowhere to go, this can be useful to process all the arguments using a loop, using one variable to reference the first argument and **shifting** until you have exhausted the arguments list.

As well as the commandline arguments there are some special builtin variables:

- ***$#*** represents the parameter count. Useful for controlling loop constructs that need to process each parameter.
- ***$@*** expands to all the parameters separated by spaces. Useful for passing all the parameters to some other function or program.
- ***$-*** expands to the flags(options) the shell was invoked with. Useful for controlling program flow based on the flags set.

- ***$$*** expands to the process id of the shell innovated to run the script. Useful for creating unique temporary filenames relative to this instantiation of the script.

3.1.3 Command Substitution

In the words of the SH manual "Command substitution allows the output of a command to be substituted in place of the command name itself". There are two ways this can be done. The first is to enclose the command like this:

```
$(command)
```

The second is to enclose the command in back quotes like this:

```
`command`
```

The command will be executed in a sub-shell environment and the standard output of the shell will replace the command substitution when the command completes.

3.1.4. Arithmetic Expansion

Arithmetic expansion is also allowed and comes in the form:

```
$((expression))
```

The value of the expression will replace the substitution. Eg:

```
echo $((1 + 3 + 4))
```

Will echo "8" to stdout

3.2 To evaluate Arithmetic Expressions:

expr-
expr writes the result of the expression on the standard output. This command is primarily intended for arithmetic and string manipulation.
Syntax:
Operators:

expr1 | *expr2*
results in the value *expr1* if *expr1* is true; otherwise it results in the value of *expr2*.
expr1 & *expr2*
results in the value of *expr1* if both expressions are true; otherwise it results in 0
expr1 <= *expr2*
expr1 < *expr2*
expr1 = *expr2*
expr1!= *expr2*
expr1 >= *expr2*
expr1 > *expr2*
If both *expr1* and *expr2* are numeric, **expr** compares them as numbers; otherwise it compares them as strings. If the comparison is true, the expression results in 1; otherwise it results in 0.
expr1 + *expr2*
expr1 - *expr2*
performs addition or subtraction on the two expressions. If either expression is not a number, **expr** exits with an error.
expr1 * *expr2*
expr1 / *expr2*
expr1 % *expr2*
performs multiplication, division, or modulus on the two expressions. If either expression is not a number, **expr** exits with an error.

3.3 Reading from Standard input

read
reading values from standard input.
Syntax: read variable_name1 variable_name2 …..
Eg: read num

Exercise:

3.1 Write a shell script to evaluate arithmetic operations.

Script1:

```
echo “enter two integer number”
read a
read b
c=`expr $a + $b`
echo “sum=$c”
c=`expr $a - $b`
echo “sub=$c”
c=`expr $a / $b`
echo “div=$c”
c=`expr $a \* $b`
echo “multiplication=$c”
c=`expr $a % $b`
```

echo “remainder=$c”

3.2 Write a shell script to calculate simple interest.

Script2:

```
echo “enter the principal value, rate of interest and time period”
 read p
 read r
 read t
 si=$(expr $p \* $r \* $t / 100)
 echo “simple interset=$si”
```

3.4 Control Constructs

The flow of control within SH scripts is done via four main constructs; if...then...elif..else, do...while, for and case.

3.4.1 If..Then..Elif..Else

This construct takes the following generic form, The parts enclosed within ([) and (]) are optional:

```
if list
then list
[elif list
then list] ...
[else list]
fi
```

When a Unix command exits it exits with what is known as an *exit status*, this indicates to anyone who wants to know the degree of success the command had in performing whatever task it was supposed to do, usually when a command executes without error it terminates with an exit status of zero. An exit status of some other value would indicate that some error had occurred, the details of which would be specific to the command. The commands' manual pages detail the exit status messages that they produce.

A list is defined in the SH as "a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters.", hence in the generic definition of the *if* above the list will determine which of the execution paths the script takes. For example, there is a command called **test** on Unix which evaluates an expression and if it evaluates to true will return zero and will return one otherwise, this is how we can test conditions in the *list* part(s) of the *if* construct because **test** is a command.

We do not actually have to type the **test** command directly into the *list* to use it, it can be implied by encasing the test case within ([) and (]) characters.

Numerical Comparision

Symbol	Description	Usage
-eq	Equals	$A -eq $B
-ne	Not equal	$A -ne $B
-lt	Less than	$A –le $B
-le	Less than or equal to	$A -le $B
-gt	Greater than	$A -ge $B
-ge	Greater than or equal to	$A –ge $B

String Comparision

Symbol	Description	Usage
=	Equals	$A = $B
!=	Not Equal	$A != $B
-n	Not a null string	-n $str
-z	Null String	-z $str

Testing For File

Symbol	Description	Usage
-r	Read Permission	-r $filename
-w	Write Permission	-w $filename
-x	Execution Permission	-x $filename
-f	File Exists	-f $filename
-d	Directory Exits	-d $filename
-c	Special Character File	-c $filename
-b	Block Special File	-b $filename
-s	Size of File is not zero	-s $filename

3.4.2 Do...While

The *Do...While* takes the following generic form:

```
while list
do list
done
```

In the words of the SH manual "The two lists are executed repeatedly while the exit status of the first list is zero." there is a variation on this that uses until in place of while which executes *until* the exit status of the first list is zero

3.4.3 For

The syntax of the for command is:

```
for variable in word ...
do list
done
```

The SH manual states “The words are expanded, and then the list is executed repeatedly with the variable set to each word in turn.”. A word is essentially some other variable that contains a list of values of some sort, the *for* construct assigns each of the values in the word to variable and then variable can be used within the body of the construct, upon completion of the body variable will be assigned the next value in word until there are no more values in word.

3.4.4 Case

The case construct has the following syntax:

```
case word in
pattern) list ;;
...

esac
```

An example of this should make things clearer:

```
!#/bin/sh
case $1
in
1) echo 'First Choice';;
2) echo 'Second Choice';;
*) echo 'Other Choice';;
esac
```

"1", "2" and "*" are patterns, word is compared to each pattern and if a match is found the body of the corresponding pattern is executed, we have used "*" to represent everything, since this is checked last we will still catch "1" and "2" because

they are checked first. In our example word is "$1", the first parameter, hence if the script is ran with the argument "1" it will output "First Choice", "2" "Second Choice" and anything else "Other Choice". In this example we compared against numbers (essentially still a string comparison however) but the pattern can be more complex,

3.4.5 Functions

The syntax of an SH function is defined as follows:

```
name ( ) command
```

It is usually laid out like this:

```
name() {
commands
}
```

A function will return with a default exit status of zero, one can return different exit status' by using the notation return *exit status*. Variables can be defined locally within a function using local *name=value*. The example below shows the use of a user defined increment function:

Example: Increment Function Example

```
#!/bin/sh
inc()
{                    # The increment is defined first so we can use it
        echo $(($1 + $2))
# We echo the result of the first parameter plus the second parameter
}

                     # We check to see that all the command line
arguments are present
if [ "$1" "" ] || [ "$2" = "" ] || [ "$3" = "" ]
then
  echo USAGE:
  echo "   counter startvalue incrementvalue endvalue"
else
  count=$1                  # Rename are variables with clearer names
  value=$2
  end=$3
  while [ $count -lt $end ]     # Loop while count is less than end
  do
```

```
    echo $count
    count=$(inc $count $value)
                        # Call increment with count and value as parameters
  done                   # so that count is incremented by value
fi
```

Exercise:

3.3 Write a shell Script to determine largest among three integer number.

Script3:

```
echo "enter three integer number"
read a
read b
read c
if [ $a –ge $b ]
then
        if [ $a –ge $c ]
        then
                echo "$a is largest number"
        else
                echo "$c is largest number"
        fi
elif [ $b –ge $c ]
then
        echo "$b is largest number"
else
        echo "$c is largest number"
fi
```

3.4 Write a shell script to determine a given year is leap year or not.

Script4:

```
echo "enter any year"
read y
if [ $(expr $y % 100) –eq 0 ]
then
        if [$(expr $y % 400) –eq 0 ]
        then
                echo "$y is leap year"
        else
                echo "$y is leap year"
        fi
elif [ $(expr $y % 4 ) –eq 0 ]
then
        echo "$y is leap year"
else
```

```
        echo "$y is leap year"
fi
```

3.5 Write a shell script to print multiplication table of given number using while statement.

Script5:

```
echo "enter any num"
read n
i=1;
while [ $i –le $n ]
do
        m=$(expr $n \* $i);
        echo "$n * $i = $m"
        i=$(expr $i + 1);
done
```

3.6 Write a shell script to compare two string.

Script6:

```
echo "enter two string"
read a
read b
if [ -z $a ]
then
        echo " First String is empty: Null String"
fi
 if [ -z $b ]
then
        echo " First String is empty: Null String"
fi
if  [ $a = $b ]
then
        echo "Strings are equal: strings Matched"
else
        echo "Strings are not equal: Strings not match"
fi
```

3.7 Write a shell script to read and check the directory exists or not, if not make directory.

Script7:

```
echo "enter name of directory"
 read dir
if [ -d $dir ]
then
        echo "Directory $dir Exits!"
else
        mkdir $dir
```

```
fi
```

3.8 Write a shell script to read and check the directory exists or not, if not make file.

Script7:

```
echo “enter name of file”
 read filename
if [ -f $filename ]
then
        echo “File $filename Exits!”
else
        touch $filename
fi
```

3.9 Write a shell script to implement menu driven program to perform all arithmetic operation using case statement.

Script9:

```
echo “enter two integer values”
read a
read b
echo –e “Menu \n 1 for Addition \n 2 for Substraction \n 3 for Multiplication \n 4 for Division \n 5 for Remainder”
echo “enter choice”
read ch
case $ch in
1)  echo “Sum=$(expr $a + $b)”;;
2)  echo “Substraction=$(expr $a - $b)”;;
3)  echo “Multiplication=$(expr $a \* $b)”;;
4)  echo “Division=$(expr $a / $b)”;;
5)  echo “Remainder=$(expr $a % $b)”;;
6)  echo “invalid Choice:Try Again!”
esac
```

3.10 Write a shell script to do:

(i). display list of directory contents
(ii). Name of current directory
(iii). Who is logged on
(iv). Long listing of directory contents
according to choice of user.

Script10:

```
echo –e “Menu \n 1 for listing directory content \n 2 for print name of current directory \n 3 for Show who is logged on \n 4 Show directory content using long listing format  “
echo “enter your choice “
read ch
case $ch in
1)  ls;;
```

```
2) pwd;;
3) who;;
4) ls –l;;
*) echo “Invalid Choice: Try Again!!”
esac
```

3.11 Write a shell script to print following pattern.

```
*
* *
* * *
* * * *
Script11 :
echo “ enter number of rows”
read n
i=1
while [ $i –le $n ]
do
        j=1
        while [ $j –le $i ]
        do
                echo –n “*”
                j=$(expr $j + 1)
        done
        echo
        i=$(expr $i + 1)
done
```

3.12 Write a shell script to read the file word by word with serial number.

```
Script12:
i=1
for line in $( cat file1)
do
        echo –e “$i \t ”
        echo –e  “$line \n”
        i=$(expr $i + 1 )
done
```

3.13 Write a shell script to read the file word by word with serial number.

```
Script12:
i=1
while read line
do
        echo –e “$i \t ”
        echo –e  “$line \n”
        i=$(expr $i + 1 )
done  <  file
```

3.14 Write a shell script to do

(i). counting number of user logged in
(ii). Printing column list of your current directory
(iii). Running your job after logging out.

Script12:

```
echo " Number of users logged in"
i=0
who > file   # result of who command  is appended to file
while  read line
do
        i=$(expr $i + 1)
done <  file
echo " $i"
i=1
for line in $( cat file)
do
        echo –e "$i \t "
        echo –e  "$line \n"
        i=$(expr $i + 1 )
done
nohup sort file
```

3.5 running job after logout

nohup

used to continue runing job / task after running out and appending output to nohup.out output file by default.

Usage: nohup [command_name]

3.6 Important Linux commands

3.6.1 date