MODULE - 2

Syllabus :
File attributes and permissions: The ls command with options. Changing file permissions: the relative and absolute permissions changing methods. Recursively changing file permissions. Directory permissions.
The shells interpretive cycle: Wild cards. Removing the special meanings of wild cards. Three standard files and redirection. **Connecting commands:** Pipe. Basic and Extended regular expressions. The grep, egrep. Typical examples involving different regular expressions.
Shell programming: Ordinary and environment variables. The .profile. Read and readonly commands. Command line arguments. exit and exit status of a command. Logical operators for conditional execution. The test command and its shortcut. The if, while, for and case control statements. The set and shift commands and handling positional parameters. The here (<<) document and trap command. Simple shell program examples.

Topics from chapters 6, 8 and 14 of text book 1

Text Book 1: Sumitabha Das., Unix Concepts and Applications., 4th Edition., Tata McGraw Hill

Chapter 6 Basic File Attributes

The UNIX file system allows the user to access other files not belonging to them and without infringing on security. A file has a number of attributes (properties) that are stored in the inode. In this chapter, we discuss,

- ls –l to display file attributes (properties)
- Listing of a specific directory
- Ownership and group ownership
- Different file permissions

Listing File Attributes

ls command is used to obtain a list of all filenames in the current directory. The output in UNIX lingo is often referred to as the listing. Sometimes we combine this option with other options for displaying other attributes, or ordering the list in a different sequence. ls look up the file's inode to fetch its attributes. It lists seven attributes of all files in the current directory and they are:

- File type and Permissions
- Links
- Ownership
- Group ownership
- File size

- Last Modification date and time
- File name

The file type and its permissions are associated with each file. Links indicate the number of file names maintained by the system. This does not mean that there are so many copies of the file. File is created by the owner. Every user is attached to a group owner. File size in bytes is displayed. Last modification time is the next field. If you change only the permissions or ownership of the file, the modification time remains unchanged. In the last field, it displays the file name.

For example,

```
$ ls –l
total 72
-rw-r--r--    1  kumar  metal  19514  may  10  13:45    chap01
-rw-r--r--    1  kumar  metal   4174  may  10  15:01    chap02
-rw-rw-rw-    1  kumar  metal     84  feb  12  12:30    dept.lst
-rw-r--r--    1  kumar  metal   9156  mar  12   1999    genie.sh
drwxr-xr-x    2  kumar  metal    512  may   9  10:31    helpdir
drwxr-xr-x    2  kumar  metal    512  may   9  09:57    progs
```

Listing Directory Attributes

ls -d will not list all subdirectories in the current directory
For example,

```
$ ls –ld helpdir progs
drwxr-xr-x  2  kumar  metal    512  may   9   10:31 helpdir
drwxr-xr-x  2  kumar  metal    512  may   9   09:57 progs
```

Directories are easily identified in the listing by the first character of the first column, which here shows a d. The significance of the attributes of a directory differs a good deal from an ordinary file. To see the attributes of a directory rather than the files contained in it, use ls –ld with the directory name. Note that simply using ls –d will not list all subdirectories in the current directory. Strange though it may seem, ls has no option to list only directories.

File Ownership

When you create a file, you become its owner. Every owner is attached to a group owner. Several users may belong to a single group, but the privileges of the group are set by the owner of the file and not by the group members. When the system administrator creates a user account, he has to assign these parameters to the user:

The user-id (UID) – both its name and numeric representation
The group-id (GID) – both its name and numeric representation

File Permissions

UNIX follows a three-tiered file protection system that determines a file's access rights. It is displayed in the following format:

Filetype owner (rwx) groupowner (rwx) others (rwx)

For Example:

-rwxr-xr-- 1 kumar metal 20500 may 10 19:21 chap02

r w x r - x r - -

owner/user group owner others

The first group has all three permissions. The file is readable, writable and executable by the owner of the file. The second group has a hyphen in the middle slot, which indicates the absence of write permission by the group owner of the file. The third group has the write and execute bits absent. This set of permissions is applicable to others.

You can set different permissions for the three categories of users – owner, group and others. It's important that you understand them because a little learning here can be a dangerous thing. Faulty file permission is a sure recipe for disaster

Changing File Permissions

A file or a directory is created with a default set of permissions, which can be determined by umask. Let us assume that the file permission for the created file is -rw-r--r--. Using **chmod** command, we can change the file permissions and allow the owner to execute his file. The command can be used in two ways:

In a relative manner by specifying the changes to the current permissions
In an absolute manner by specifying the final permissions

Relative Permissions

chmod only changes the permissions specified in the command line and leaves the other permissions unchanged. Its syntax is:

chmod category operation permission filename(s)

chmod takes an expression as its argument which contains:
user category (user, group, others)
operation to be performed (assign or remove a permission)

type of permission (read, write, execute)

Category	**operation**	**permission**
u - user	+ assign	r - read
g - group	- remove	w - write
o - others	= absolute	x - execute
a - all (ugo)		

Let us discuss some examples:

Initially,

-rw-r--r-- 1 kumar metal 1906 sep 23:38 xstart

$chmod u+x xstart

-rwxr--r-- 1 kumar metal 1906 sep 23:38 xstart

The command assigns (+) execute (x) permission to the user (u), other permissions remain unchanged.

$chmod ugo+x xstart or
$chmod a+x xstart or
$chmod +x xstart

-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

chmod accepts multiple file names in command line

$chmod u+x note note1 note3

Let initially,

-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

$chmod go-r xstart

Then, it becomes

-rwx--x--x 1 kumar metal 1906 sep 23:38 xstart

Absolute Permissions

Here, we need not to know the current file permissions. We can set all nine permissions explicitly. A string of three octal digits is used as an expression. The permission can be represented by one octal digit for each category. For each category, we add octal digits. If we represent the permissions of each category by one octal digit, this is how the permission can be represented:

- Read permission – 4 (octal 100)
- Write permission – 2 (octal 010)
- Execute permission – 1 (octal 001)

Octal	Permissions	Significance
0	- - -	no permissions
1	- - x	execute only
2	- w -	write only
3	- w x	write and execute
4	r - -	read only
5	r - x	read and execute
6	r w -	read and write
7	r w x	read, write and execute

We have three categories and three permissions for each category, so three octal digits can describe a file's permissions completely. The most significant digit represents user and the least one represents others. chmod can use this three-digit string as the expression.

Using relative permission, we have,

$chmod a+rw xstart

Using absolute permission, we have,

$chmod 666 xstart

$chmod 644 xstart

$chmod 761 xstart

will assign all permissions to the owner, read and write permissions for the group and only execute permission to the others.

777 signify all permissions for all categories, but still we can prevent a file from being deleted. 000 signifies absence of all permissions for all categories, but still we can delete a file. It is the directory permissions that determine whether a file can be deleted or not. Only owner can change the file permissions. User cannot change other user's file's permissions. But the system administrator can do anything.

The Security Implications

Let the default permission for the file xstart is

-rw-r--r--

$chmod u-rw, go-r xstart or

$chmod 000 xstart

This is simply useless but still the user can delete this file
On the other hand,

$chmod a+rwx xstart

$chmod 777 xstart

-rwxrwxrwx

The UNIX system by default, never allows this situation as you can never have a secure system. Hence, directory permissions also play a very vital role here

We can use chmod Recursively.

$chmod -R a+x shell_scripts

This makes all the files and subdirectories found in the shell_scripts directory, executable by all users. When you know the shell meta characters well, you will appreciate that the * doesn't match filenames beginning with a dot. The dot is generally a safer but note that both commands change the permissions of directories also.

Directory Permissions

It is possible that a file cannot be accessed even though it has read permission, and can be removed even when it is write protected. The default permissions of a directory are,

rwxr-xr-x (755)

A directory must never be writable by group and others

Example:

$mkdir c_progs

$ls –ld c_progs

drwxr-xr-x 2 kumar metal 512 may 9 09:57 c_progs

If a directory has write permission for group and others also, be assured that every user can remove every file in the directory. As a rule, you must not make directories universally writable unless you have definite reasons to do so.

Changing File Ownership

Usually, on BSD and AT&T systems, there are two commands meant to change the ownership of a file or directory. Let kumar be the owner and metal be the group owner. If sharma copies a file of kumar, then sharma will become its owner and he can manipulate the attributes

chown changing file owner and chgrp changing group owner

On BSD, only system administrator can use chown
On other systems, only the owner can change both

chown

Changing ownership requires superuser permission, so use **su** command

```
$ls -l note

-rwxr----x      1 kumar  metal  347  may  10  20:30  note

$chown sharma note; ls -l note

-rwxr----x      1 sharma  metal  347  may  10  20:30  note
```

Once ownership of the file has been given away to sharma, the user file permissions that previously applied to Kumar now apply to sharma. Thus, Kumar can no longer edit *note* since there is no write privilege for group and others. He can not get back the ownership either. But he can copy the file to his own directory, in which case he becomes the owner of the copy.

chgrp

This command changes the file's group owner. No superuser permission is required.

```
$ls –l dept.lst

-rw-r--r--      1  kumar  metal  139  jun  8  16:43  dept.lst

$chgrp dba dept.lst; ls –l dept.lst

-rw-r--r--      1  kumar  dba  139  jun  8  16:43  dept.lst
```

In this chapter we considered two important file attributes – permissions and ownership. After we complete the first round of discussions related to files, we will take up the other file attributes.

Chapter 8 The Shell

Introduction

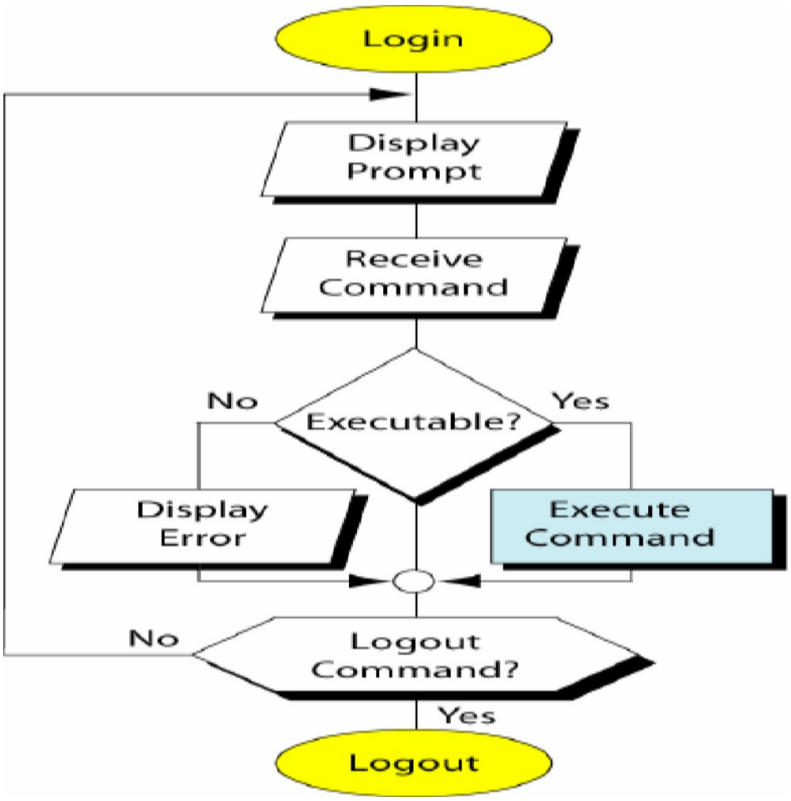
In this chapter we will look at one of the major component of UNIX architecture – The Shell. Shell acts as both a command interpreter as well as a programming facility. We will look at the interpretive nature of the shell in this chapter.

Objectives

- The Shell and its interpretive cycle
- Pattern Matching – The wild-cards
- Escaping and Quoting
- Redirection – The three standard files
- Filters – Using both standard input and standard output
- /dev/null and /dev/tty – The two special files
- Pipes
- tee – Creating a tee
- Command Substitution
- Shell Variables

1. The shell and its interpretive cycle

The shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to ***command.com*** in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files. The original shell was the Bourne shell, ***sh***. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.



Numerous other shells are available. Some of the more well known of these may be on your Unix system: the Korn shell, ***ksh***, by David Korn, C shell, ***csh***, by Bill Joy and the Bourne Again SHell, ***bash***, from the Free Software Foundations GNU project, both based on ***sh***, the T-C shell, ***tcsh***, and the extended C shell, ***cshe***, both based on ***csh***.

Even though the shell appears not to be doing anything meaningful when there is no activity at the terminal, it swings into action the moment you key in something.

The following activities are typically performed by the shell in its interpretive cycle:

- The shell issues the prompt and waits for you to enter a command.
- After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like the * in rm *) to recreate a simplified command line.
- It then passes on the command line to the kernel for execution.
- The shell waits for the command to complete and normally can't do any work while the command is running.
- After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. You are free to enter another command.

2. Pattern Matching – The Wild-Cards

A pattern is framed using ordinary characters and a metacharacter (like *) using well-defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed.

The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards. The following table lists them:

Wild-Card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of characters x and x
[!ijk]	A single character that is not an i,j or k (Not in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters x and x (Not in C Shell)
{pat1,pat2… }	Pat1, pat2, etc. (Not in Bourne shell)

Examples:

To list all files that begin with *chap*, use

$ ls chap*

To list all files whose filenames are six character long and start with chap, use

$ ls chap??

Note: Both * and ? operate with some restrictions. for example, the * doesn't match all files beginning with a . (dot) ot the / of a pathname. If you wish to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

$ ls .???*

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly.

Similarly, these characters don't match the / in a pathname. So, you cannot use

$ cd /usr?local to change to /usr/local.

The character class

You can frame more restrictive patterns with the character class. The character class comprises a set of characters enclosed by the rectangular brackets, [and], but it matches a single character in the class. The pattern [abd] is character class, and it matches a single character – an a,b or d.

Examples:

$ls chap0[124] Matches chap01, chap02, chap04 and lists if found.

$ ls chap[x-z] Matches chapx, chapy, chapz and lists if found.

You can negate a character class to reverse a matching criteria. For example,

- To match all filenames with a single-character extension but not the .c ot .o files, use *.[!co]
- To match all filenames that don't begin with an alphabetic character, use [!a-zA-Z]*

Matching totally dissimilar patterns

This feature is not available in the Bourne shell. To copy all the C and Java source programs from another directory, we can delimit the patterns with a comma and then put curly braces around them.

$ cp $HOME/prog_sources/*.{c,java} .

The Bourne shell requires two separate invocations of cp to do this job.

$ cp /home/srm/{project,html,scripts/* .

The above command copies all files from three directories (project, html and scripts) to the current directory.

3. Escaping and Quoting (Removing the special meanings of wild cards)

Escaping is providing a \ (backslash) before the wild-card to **remove (escape)** its special meaning.

For instance, if we have a file whose filename is chap* (Remember a file in UNIX can be names with virtually any character except the / and null), to remove the file, it is dangerous to give command as *rm chap**, as it will remove all files beginning with chap. Hence to suppress the special meaning of *, use the command *rm chap**

To list the contents of the file chap0[1-3], use

$ cat chap0\[1-3\]

A filename can contain a whitespace character also. Hence to remove a file named

My Documend.doc, which has a space embedded, a similar reasoning should be followed:

$ rm My\ Document.doc

Quoting is enclosing the wild-card, or even the entire pattern, within quotes. Anything within these quotes (barring a few exceptions) are left alone by the shell and not interpreted.

When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

Examples:

$ rm ‘chap*’ Removes fil *chap**

$ rm “My Document.doc” Removes file *My Document.doc*

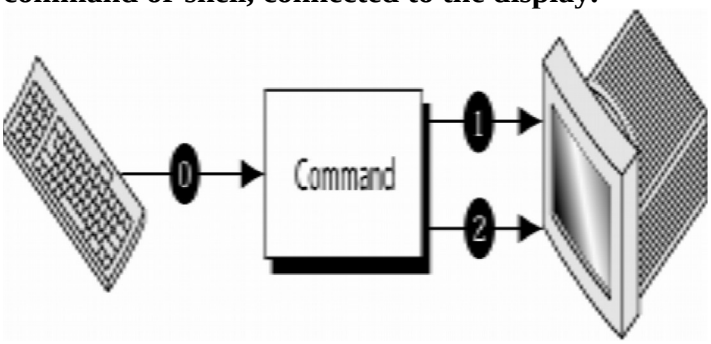
4. Redirection : The three standard files

The shell associates three files with the terminal – two for display and one for the keyboard. These files are streams of characters which many commands see as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device:

Standard input: The file (stream) representing input, connected to the keyboard.

Standard output: The file (stream) representing output, connected to the display.

Standard error: The file (stream) representing error messages that emanate from the command or shell, connected to the display.



The standard input can represent three input sources:

1. The keyboard, the default source.
2. A file using redirection with the < symbol.
3. Another program using a pipeline.

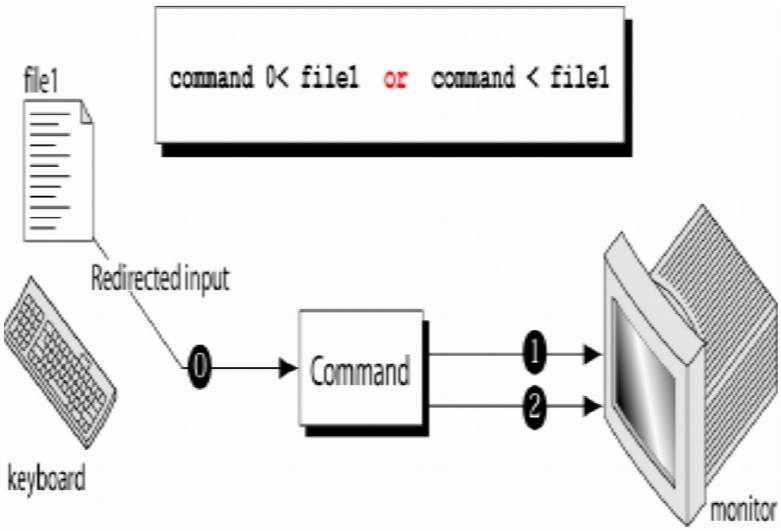
The standard output can represent three possible destinations:

1. The terminal, the default destination.
2. A file using the redirection symbols > and >>.
3. As input to another program using a pipeline.

A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first three slots are generally allocated to the three standard streams as,

0 – Standard input
1 – Standard output
2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.
Examples:



Assuming file2 doesn't exist, the following command redirects the standard output to file *myOutput* and the standard error to file *myError*.

$ ls –l file1 file2 1>myOutput 2>myError

To redirect both standard output and standard error to a single file use:

$ ls –l file1 file2 1>| myOutput 2>| myError OR
$ ls –l file1 file2 1> myOutput 2>& 1

5. Filters: Using both standard input and standard output

UNIX commands can be grouped into four categories viz.,

1. Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2. Commands like ls, pwd, who etc. don't read standard input but they write to standard output.
3. Commands like lp that read standard input but don't write to standard output.
4. Commands like cat, wc, cmp etc. that use both standard input and standard output.

Commands in the fourth category are called filters. Note that filters can also read directly from files whose names are provided as arguments.

Example: To perform arithmetic calculations that are specified as expressions in input file calc.txt and redirect the output to a file result.txt, use

calc. txt

$cat >calc.txt
3*4
9-6
8/2
^d

$ bc < calc.txt > result.txt

$cat result.txt
12 3 4

7. Pipes: Connecting Commands

With piping, the output of a command can be used as input (piped) to a subsequent command.

$ command1 | command2

Output from command1 is piped into input for command2.

This is equivalent to, but more efficient than:

$ command1 > temp
$ command2 < temp
$ rm temp

Examples

$ ls -al | more
$ who | sort | lpr

When a command needs to be ignorant of its source

If we wish to find total size of all C programs contained in the working directory, we can use the command,

$ wc –c *.c

However, it also shows the usage for each file(size of each file). We are not interested in individual statistics, but a single figure representing the total size. To be able to do that, we must make wc ignorant of its input source. We can do that by feeding the concatenated output stream of all the .c files to wc –c as its input:

$ cat *.c | wc –c

Basic and Extended regular expressions

We often need to search a file for a pattern, either to see the lines containing (or not containing) it or to have it replaced with something else. This chapter discusses two important filters that are especially suited for these tasks – grep and sed. grep takes care of all search requirements we may have. sed goes further and can even manipulate the individual characters in a line. In fact sed can do several things.

grep – searching for a pattern

It scans the file / input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs. It's a command from a special family in UNIX for handling search requirements.

grep ***options pattern filename(s)***

$grep "sales" emp.lst

will display lines containing sales from the file emp.lst. Patterns with and without quotes is possible. It's generally safe to quote the pattern. Quote is mandatory when pattern involves more than one word. It returns the prompt in case the pattern can't be located.

$grep president emp.lst

When grep is used with multiple filenames, it displays the filenames along with the output.

$grep "director" emp1.lst emp2.lst

Where it shows filename followed by the contents

grep options

grep is one of the most important UNIX commands, and we must know the options that POSIX requires grep to support. Linux supports all of these options.

-i	ignores case for matching
-v	doesn't display lines matching expression
-n	displays line numbers along with lines
-c	displays count of number of occurrences
-l	displays list of filenames only
-e exp	specifies expression with this option
-x	matches pattern with entire line
-f file	takes pattrens from file, one per line
-E	treats pattren as an extended RE
-F	matches multiple fixed strings

$grep -i 'agarwal' emp.lst

$grep -v 'director' emp.lst > otherlist

$wc -l otherlist will display 11 otherlist

$grep –n 'marketing' emp.lst

$grep –c 'director' emp.lst

$grep –c 'director' emp*.lst

will print filenames prefixed to the line count

$grep –l 'manager' *.lst

will display filenames *only*

$grep –e 'Agarwal' –e 'aggarwal' –e 'agrawal' emp.lst

will print matching multiple patterns

$grep –f pattern.lst emp.lst

all the above three patterns are stored in a separate file *pattern.lst*

Basic Regular Expressions (BRE) – An Introduction

It is tedious to specify each pattern separately with the -e option. grep uses an expression of a different type to match a group of similar patterns. If an expression uses meta characters, it is termed a regular expression. Some of the characters used by regular expression are also meaningful to the shell.

BRE character subset

The basic regular expression character subset uses an elaborate meta character set, overshadowing the shell's wild-cards, and can perform amazing matches.

Symbols or Expression	Matches
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg, gggg, etc.
.	A single character
.*	Nothing or any number of characters
[pqr]	A single character p, q or r
[c1-c2]	A single character withing ASCII range shown by c1 and c2
[0-9]	A digit between 0 and 9
[^pqr]	A single character which is not a p, q or r
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern pat at beginning of line
pat$	Pattern pat at end of line
^bash$	A bash as the only word in line
^$	Lines containing nothing

The character class

grep supports basic regular expressions (BRE) by default and extended regular expressions (ERE) with the –E option. A regular expression allows a group of characters enclosed within a pair of [ ], in which the match is performed for a single character in the group.

$grep "[aA]g[ar][ar]wal" emp.lst

A single pattern has matched two similar strings. The pattern [a-zA-Z0-9] matches a single alphanumeric character. When we use range, make sure that the character on the left of the hyphen has a lower ASCII value than the one on the right. Negating a class (^) (caret) can be used to negate the character class. When the character class begins with this character, all characters other than the ones grouped in the class are matched.

The *

The asterisk refers to the immediately preceding character. * indicates zero or more occurrences of the previous character.

g* nothing or g, gg, ggg, etc.

$grep “[aA]gg*[ar][ar]wal” emp.lst

Notice that we don’t require to use –e option three times to get the same output!!!!!

The dot

A dot matches a single character. The shell uses ? Character to indicate that.

.* signifies any number of characters or none

$grep “j.*saxena” emp.lst

Specifying Pattern Locations (^ and $)

Most of the regular expression characters are used for matching patterns, but there are two that can match a pattern at the beginning or end of a line. Anchoring a pattern is often necessary when it can occur in more than one place in a line, and we are interested in its occurance only at a particular location.

^	for matching at the beginning of a line
$	for matching at the end of a line

$grep “^2” emp.lst

Selects lines where emp_id starting with 2
$grep “7…$” emp.lst

Selects lines where emp_salary ranges between 7000 to 7999

$grep “^[^2]” emp.lst

Selects lines where emp_id doesn’t start with 2

When meta characters lose their meaning

It is possible that some of these special characters actually exist as part of the text. Sometimes, we need to escape these characters. For example, when looking for a pattern g*, we have to use \
To look for [, we use \[

To look for .*, we use \.*

Extended Regular Expression (ERE) and grep

If current version of grep doesn't support ERE, then use egrep but without the –E option. -E option treats pattern as an ERE.

+ matches one or more occurrences of the previous character

? Matches zero or one occurrence of the previous character

b+ matches b, bb, bbb, etc.

b? matches either a single instance of b or nothing

These characters restrict the scope of match as compared to the *

$grep –E "[aA]gg?arwal" emp.lst

?include +<stdio.h>

The ERE set

Expression	Significance
ch+	Matches one or more occurrences of character ch
ch?	Matches zero or one occurrence of character ch
exp1 | exp2	Matches exp1 or exp2
GIF | JPEG	Matches GIF or JPEG
(x1|x2)x3	Matches x1x3 or x2x3
(hard|soft)ware	Matches hardware or software

Matching multiple patterns (|, (and))

$grep –E 'sengupta|dasgupta' emp.lst

We can locate both without using –e option twice, or

$grep –E '(sen|das)gupta' emp.lst

Ordinary and environment variables

The Shell

The UNIX shell is both an interpreter as well as a scripting language. An interactive shell turns noninteractive when it executes a script.

Bourne Shell – This shell was developed by Steve Bourne. It is the original UNIX shell. It has strong programming features, but it is a weak interpreter.

C Shell – This shell was developed by Bill Joy. It has improved interpretive features, but it wasn't suitable for programming.

Korn Shell – This shell was developed by David Korn. It combines best features of the bourne and C shells. It has features like aliases, command history. But it lacks some features of the C shell.

Bash Shell – This was developed by GNU. It can be considered as a superset that combined the features of Korn and C Shells. More importantly, it conforms to POSIX shell specification.

Environment Variables

We already mentioned a couple of environment variables, such as `PATH` and `HOME`. Until now, we only saw examples in which they serve a certain purpose to the shell. But there are many other UNIX utilities that need information about you in order to do a good job.

What other information do programs need apart from paths and home directories? A lot of programs want to know about the kind of terminal you are using; this information is stored in the `TERM` variable. The shell you are using is stored in the `SHELL` variable, the operating system type in `OS` and so on. A list of all variables currently defined for your session can be viewed entering the **env** command.

The environment variables are managed by the shell. As opposed to regular shell variables, environment variables are inherited by any program you start, including another shell. New processes are assigned a copy of these variables, which they can read, modify and pass on in turn to their own child processes.

The set statement display all variables available in the current shell, but env command displays only environment variables. Note than env is an external command and runs in a child process.

There is nothing special about the environment variable names. The convention is to use uppercase letters for naming one.

The Common Environment Variables

The following table shows some of the common environment variables.

Variable name	Stored information
`HISTSIZE`	size of the shell history file in number of lines
`HOME`	path to your home directory
`HOSTNAME`	local host name
`LOGNAME`	login name

MAIL	location of your incoming mail folder
MANPATH	paths to search for man pages
PATH	search paths for commands
PS1	primary prompt
PS2	secondary prompt
PWD	present working directory
SHELL	current shell
TERM	terminal type
UID	user ID
USER	Login name of user
MAILCHECK	Mail checking interval for incoming mail
CDPATH	List of directories searched by cd when used with a non-absolute pathname

We will now describe some of the more common ones.

The command search path (PATH): The PATH variable instructs the shell about the route it should follow to locate any executable command.
Your home directory (HOME): When you log in, UNIX normally places you in a directory named after your login name. This is called the home directory or login directory. The home directory for a user is set by the system administrator while creating users (using useradd command).
mailbox location and checking (MAIL and MAILCHECK): The incoming mails for a user are generally stored at /var/mail or /var/spool/mail and this location is available in the environment variable MAIL. MAILCHECK determines how often the shell checks the file for arrival of new mail.
The prompt strings (PS1, PS2): The prompt that you normally see (the $ prompt) is the shell's primary prompt specified by PS1. PS2 specifies the secondary prompt (>). You can change the prompt by assigning a new value to these environment variables.
Shell used by the commands with shell escapes (SHELL): This environment variable specifies the login shell as well as the shell that interprets the command if preceded with a shell escape.

Variables used in Bash and Korn

The Bash and korn prompt can do much more than displaying such simple information as your user name, the name of your machine and some indication about the present working directory. Some examples are demonstrated next.

```
$ PS1='[PWD] '
[/home/srm] cd progs
[/home/srm/progs] _
```

Bash and Korn also support a *history* facility that treats a previous command as an *event* and associates it with a number. This event number is represented as !.

```
$ PS1='[!] '                $ PS1='[! $PWD] '
[42] _                      [42 /home/srm/progs] _

$ PS1="\h> "          // Host name of the machine
saturn> _
```

Aliases

Bash and korn support the use of aliases that let you assign shorthand names to frequently used commands. Aliases are defined using the alias command. Here are some typical aliases that one may like to use:

```
alias lx='/usr/bin/ls -lt'
alias l='/usr/bin/ls -l'
```

You can also use aliasing to redefine an existing command so it is always invoked with certain options. For example:

alias cp="cp –i"
alias rm="rm –i"

Note that to execute the original external command, you have to precede the command with a \. This means that you have to use \cp file1 file2 to override the alias.

The alias command with a argument displays its alias definition, if defined. The same command without any arguments displays all aliases and to unset an alias use unalias statement. To unset the alias cp, use unalias cp

2. read: MAKING SCRIPT INTERACTIVE

The read statement is the shell's internal tool for taking input from the user, i.e. making script interactive.

Example:

```
#!/bin/sh
#Sample Shell Script - simple.sh
echo "Enter Your First Name:"
read fname
echo "Enter Your Last Name:"
read lname
echo "Your First Name is: $fname"
echo " Your Last Name is: $lname"
```

Execution & Output:

```
$ chmod 777 simple.sh
$ sh simple.sh
```

Enter Your First Name: Henry
Enter Your Last Name: Ford
Your First Name is: Henry
Your Last Name is: Ford

3. USING COMMAND LINE ARGUMENTS

- Shell scripts also accept arguments from the command line.
- They can, therefore, run non-interactively and be used with redirection and pipelines.
- When arguments are specified with a shell script, they are assigned to positional parameters.
- The shell uses following parameters to handle command line arguments-

Shell parameter	Significance
$#	Number of arguments specified in command line
$0	Name of executed command
$1, $2, …	Positional parameters representing command line arguments
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string is treated as a separate arguments, same as $*

Example:

```
#!/bin/sh
#Shell Script to demonstrate command line arguments - sample.sh
echo "The Script Name is: $0"
echo "Number of arguments specified is: $#"
echo "The arguments are: $*"
echo "First Argument is: $1"
echo "Second Argument is: $2"
echo "Third Argument is: $3"
echo "Fourth Argument is: $4"
echo "The arguments are: $@"
```

Execution & Output:

```
$ sh sample.sh welcome to hit nidasoshi [Enter]
The Script Name is: sample.sh
```

Number of arguments specified is: 4
The arguments are: welcome to hit nidasoshi

First Argument is: welcome
Second Argument is: to
Third Argument is: hit
Fourth Argument is: nidasoshi
The arguments are: welcome to hit nidasoshi

4. exit AND EXIT STATUS OF COMMAND

C program and shell scripts have a lot in common, and one of them is that they both use the same command (or function in c) to terminate a program.It has the name exit in the shell and exit( ) in C.

The command is usually run with numeric arguments:

- **exit 0 #Used when everything went fine**
- **exit 1 #Used when something went wrong**

The shell offers a variable $? and a command test that evaluates a command's exit status.

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.

This parameter is set by exit's argument.

Examples:

- $ grep director emp.lst >/dev/null; echo $?

0 #Success

- $ grep director emp.lst >/dev/null; echo $?

1 #Failure – in finding pattern

5. THE LOGICAL OPERATORS && AND || - CONDITIONAL EXECUTION

The shell provides two operators that allow conditional execution – the && and | |.

Examples:

$ date && echo “Date Command Executed Successfully!”

Sun Jan 13 15:40:13 IST 2013

Date Command Executed Successfully!

$ grep 'director' emp.lst && echo “Pattern found in File!”

1234 | Henry Ford | director | Marketing | 12/12/12 | 25000

Pattern found in File!

$ grep 'manager' emp.lst | | echo “Pattern not-found in File!”
Pattern not-found in File!

7. USING test AND [ ] TO EVALUATE EXPRESSIONS

When you use if to evaluate expressions, you need the test statement because the true or false values returned by expression's can't be directly handled by if.
test uses certain operators to evaluate the condition on its right and returns either a true or false
exit status, which is then used by if for making decision.
test works in three ways:

- Compares two numbers
- Compares two strings or a single one for a null value.
- Checks a file's attributes

test doesn't display any output but simply sets the parameter $?.

Numeric Comparison

Numerical Comparison operators used by test:

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to

The numerical comparison operators used by test always begins with a - (hyphen), followed by a two-letter string, and enclosed on either side by whitespace.

Examples:

$ x=5, y=7, z=7.2

$ test $x -eq $y; echo $?

1 # Not Equal

$ test $x -lt $y; echo $?

0 #True

$ test $y -eq $z

0 #True- 7.2 is equal to 7

The last example proves that numeric comparison is restricted to integers only.

The [ ] is used as shorthand for test.

Hence, above example may be re-written as test

$x -eq $y or [$x -eq $y] #Both are equivalent

6. THE if CONDITIONAL

The if statement makes two-way decisions depending on the fulfillment of a certain condition.

In the shell, the statement uses the following forms

if command is successful
then
  execute commands
else
  execute commands
fi
	If command is successful
	then
	  execute commands
	fi
		If command is successful
		then
		  execute commands
		elif command is successful
		then
		  execute commands
		else
		  execute commands
		fi

Form 1	Form 2	Form 3

Example:

#!/bin/sh

#Shell script to illustrate if conditional

```
if grep 'director' emp.lst >/dev/null
then
echo “Pattern found in File!”
else
echo “Pattern not-found in File!”
fi
```

String Comparison

test can be used to compare strings with yet another set of operators. The below table shows string tests used by test-

Test	True if
s1 = s2	String s1 = s2
s1 != s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not a null string
s1 == s2	String s1 = s2 (Korn and Bash only)

Example:

```
#!/bin/sh
#Shell script to illustrate string comparison using test – strcmp.sh
if [ $# -eq 0 ]; then
echo "Enter Your Name: \c"; read name;
[ -z $name ]; then
echo "You have not entered your name! "; exit 1;
else
echo "Your Name From Input line is : $name "; exit 1;
fi
else
echo "Your Name From Command line is : $1 " ;
```

```
fi
```

Execution & Output:

```
$ chmod 777 strcmp.sh
$ sh strcmp.sh Henry
Your Name From Command line is : Henry
$ sh strcmp.sh
Enter Your Name: Smith [Enter]
Your Name From Input line is : Smith
$ sh strcmp.sh
Enter Your Name: [Enter]
You have not entered your name!
```

8. THE case CONDITIONAL

The case statement is similar to switch statement in C.
The statement matches an expression for more than one alternative, and permit multi-way
branching.
The general syntax of the case statement is as follows:

```
case expression in
pattern1) command1 ;;
pattern2) command2 ;;
pattern3) command3 ;;
……....
esac
```

Example:

```
#!/bin/sh
#Shell script to illustrate CASE conditional – menu.sh
echo "\t MENU\n 1. List of files\n 2. Today's Date\n 3. Users of System\n 4. Quit\
n";
echo "Enter your option: \c";
read choice
case "$choice" in
1) ls -l ;;
2) date ;;
3) who ;;
4) exit ;;
```

```
*) echo "Invalid Option!"
esac
```

Execution & Output:

```
$ chmod 777 menu.sh
$ sh menu.sh
MENU
1. List of files
2. Today's Date
3. Users of System
4. Quit
Enter your option: 2
Sun Jan 13 15:40:13 IST 2013
```

10. while: LOOPING

The while statement should be quite familiar to many programmers.
It repeatedly performs a set of instructions until the control commands returns a true exit status.

The general syntax of this command is as follows:

```
while condition is true
do
commands
done
```

The commands enclosed by do and done are executed repeatedly as long as condition remains true.

Example:

```
#!/bin/sh
#Shell script to illustrate while loop – while.sh
answer=y;
while [ "$answer" = "y" ]
do
echo "Enter Branch Code and Name: \c"
read code name #Read both together
echo "$code|$name" >> newlist #Append a line to newlist
echo "Enter anymore (y/n)? \c"
read anymore
```

```
case $anymore in
y*|Y*) answer=y ;; #Also accepts yes, YES etc.
n*|N*) answer=n ;; #Also accepts no, NO etc.
*) answer=n ;; #Any other reply means no
esac
done
```

Execution & Output:

```
$ chmod 777 while.sh
$ sh while.sh
Enter Branch Code and Name: CS COMPUTER [Enter]
Enter anymore (y/n)? y [Enter]
Enter Branch Code and Name: EC ELECTRONICS [Enter]
Enter anymore (y/n)? n [Enter]
$ cat newlist
CS|COMPUTER
EC|ELECTRONICS
```

11. for: LOOPING WITH A LIST

The shell's for loop differs in structure from the ones used in other programming language.

Unlike while and until, for doesn't test a condition, but uses a list instead.

The general syntax of for loop is as follows

for
variable in list
do
commands
done

The loop body also uses the keywords do and done, but the additional parameters here are variable and list.

Each whitespace-separated word in list is assigned to variable in turn, and commands are executed until list is exhausted.

A simple example can help you understand for loop better:

Example:

```
#!/bin/sh
#Shell script to illustrate use of for loop – forloop.sh
for file in chap1 chap2 chap3 chap4
do
cp $file ${file}.bak
echo “$file copied to $file.bak”
```

done

Execution & Output:
$ chmod 777 forloop.sh
$ sh forloop.sh
chap1 copied to chap1.bak
chap2 copied to chap2.bak
chap3 copied to chap3.bak
chap4 copied to chap4.bak

Possible sources of the list:
1. list from variables - $ for var in $x $y $z
2. list from command substitution- $ for var in `cat foo`
3. list from wild-cards- $ for file in *.htm *.html
4. list from positional parameters- $ for var in "$@"

12. set AND shift: MANIPULATING THE POSITIONAL PARAMETERS

set: Set the positional parameters
set assigns its arguments to the positional parameters $1, $2 and so on.
This feature is especially useful for picking up individual fields from the output of a program.

Example:
$ set `date` #Output of date command assigned to positional parameters $1, $2 & so on.
$ echo $*
Sun Jan 13 15:40:13 IST 2013
$ echo "The date today is $2 $3 $6"
The date today is Jan 13 2013

shift: Shifting Arguments Left
shift transfers the contents of a positional parameters to its immediate lower numbered one.
This is done as many times as the statement is called.

Example:
$ set `date`
$ echo $*

Sun Jan 13 15:40:13 IST 2013
$ echo $1 $2 $3
Sun Jan 13
$ shift #Shifts 1 place
Jan 13 15:40:13
$ echo $1 $2 $3
$ shift 2 #Shifts 2 places
$ echo $1 $2 $3
15:40:13 IST 2013

The Here Document (<<)

The shell uses the << symbol to read data from the same file containing the script. This is referred to as a here document, signifying that the data is here rather than in aspirate file. Any command using standard input can slo take input from a here document.

Example:

```
mailx kumar << MARK
Your program for printing the invoices has been
executed on `date`.Check the print queue
The updated file is
$flname MARK
```

The string (MARK) is delimiter. The shell treats every line following the command and delimited by MARK as input to the command. Kumar at the other end will see three lines of message text with the date inserted by command. The word MARK itself doesn't show up.

Using Here Document with Interactive Programs:

A shell script can be made to work non-interactively by supplying inputs through here
document.

Example:

```
$ search.sh <<
END > director
>emp.lst
>END
```

Output:

Enter the pattern to be searched: Enter the file to be used: Searching for director from file
emp.lst

9876 Jai Sharma Director Productions
2356 Rohit Director Sales
Selected records shown above.
The script search.sh will run non-interactively and display the lines containing "director" in the file emp.lst.

trap: interrupting a Program

Normally, the shell scripts terminate whenever the interrupt key is pressed. It is not a good programming practice because a lot of temporary files will be stored on disk. The trap statement lets you do the things you want to do when a script receives a signal. The trap statement is normally placed at the beginning of the shell script and uses two lists:

trap 'command_list' signal_list
When a script is sent any of the signals in signal_list, trap executes the commands in command_list. The signal list can contain the integer values or names (without SIG prefix) of one or more signals – the ones used with the kill command.
Example: To remove all temporary files named after the PID number of the shell:
trap 'rm $$* ; echo "Program Interrupted" ; exit' HUP INT TERM trap is a signal handler. It first removes all files expanded from $$*, echoes a message and finally terminates the script when signals SIGHUP (1), SIGINT (2) or SIGTERM(15) are sent to the shell process running the script.
A script can also be made to ignore the signals by using a null command list.
Example:
trap '' 1 2 15

Programs 1)

```
#!/bin/sh
IFS="|"
While echo "enter dept code:\c"; do
Read dcode
Set -- `grep "^$dcode"<<limit
01|ISE|22
02|CSE|45
03|ECE|25
04|TCE|58
limit`
Case $# in
```

```
3) echo “dept name :$2 \n empid:$
3\n” *) echo “invalid
code”;continue
esac
done
```

Output:

```
$valcode.sh
Enter dept code:88
Invalid code
Enter dept code:02
Dept name : CSE
Emp-id :45
Enter dept code:<ctrl-c>
```

Program 2)

```
#!/bin/sh
x=1
While [$x –le 10];do
echo “$x”
x=`expr $x+1`
done
#!/bin/sh
sum=0
for I in “$@” do
echo “$I”
sum=`expr $sum + $I`
done
Echo “sum is $sum”
```

Program 3)

```
#!/bin/sh
sum=0
for I in `cat list`; do
echo “string is $I”
x= `expr “$I”:’.*’`
Echo “length is $x”
Done
```

Program 4)
This is a non-recursive shell script that accepts any number of arguments and prints them in a reverse order.
For example if A B C are entered then output is C B A.

```
#!/bin/sh
if [ $# -lt 2 ]; then
echo "please enter 2 or more arguments"
exit
fi
for x in $@
do
y=$x" "$y
done echo
"$y"
```

Run1:
[root@localhost shellprgms]# sh sh1a.sh 1 2 3 4 5 6 7
7 6 5 4 3 2 1
Run2: [root@localhost shellprgms]# sh ps1a.sh this is an argument
argument an is this

Program 5)The following shell script to accept 2 file names checks if the permission for these files are identical and if they are not identical outputs each filename followed by permission.

```
#!/bin/sh
if [ $# -lt 2 ]
then
echo "invalid number of arguments"
exit
fi
str1=`ls -l $1|cut -c 2-10`
str2=`ls -l $2|cut -c 2-10`
if [ "$str1" = "$str2" ]
then
echo "the file permissions are the same: $str1"
else
```

```
echo " Different file permissions "
echo -e "file permission for $1 is $str1\nfile permission for $2 is $str2"
fi
```

Run1:

```
[root@localhost shellprgms]# sh 2a.sh ab.c xy.c
file permission for ab.c is rw-r--r--
file permission for xy.c is rwxr-xr-x
Run2:
[root@localhost shellprgms]# chmod +x ab.c
[root@localhost shellprgms]# sh 2a.sh ab.c xy.c
the file permissions are the same: rwxr-xr-x
```

Program 6)This shell function that takes a valid directory name as an argument and recursively descends all the subdirectories, finds the maximum length of any file in that hierarchy and writes this maximum value to the standard output.

```
#!/bin/sh
if [ $# -gt 2 ]
then
echo "usage sh flname dir"
exit
fi
if [ -d $1 ]
then
ls -lR $1|grep -v ^d|cut -c 34-43,56-69|sort -n|tail -1>fn1
echo "file name is `cut -c 10- fn1`"
echo " the size is `cut -c -9 fn1`"
else
echo "invalid dir name"
fi
```

Run1:

```
[root@localhost shellprgms]# sh 3a.sh
file name is a.out
the size is 12172
```

Program 7)This shell script that accepts valid log-in names as arguments and prints their corresponding home directories. If no arguments are specified, print a suitable error message.

```
if [ $# -lt 1 ]
then
echo " Invlaid Arguments..... "
exit
fi
for x in "$@"
do
grep -w "^$x" /etc/passwd | cut -d ":" -f 1,6
done
```

Run1:

```
[root@localhost shellprgms]# sh 4a.sh root
root:/root
```

Run2:

```
[root@localhost shellprgms]# sh 4a.sh
Invalid Arguments.....
```

Program 8) This shell script finds and displays all the links of a file specified as the first argument to the script. The second argument, which is optional, can be used to specify the directory in which the search is to begin. If this second argument is not present .the search is to begin in current working directory.

```
#!/bin/bash
if [ $# -eq 0 ]
then
echo "Usage:sh 8a.sh[file1] [dir1(optional)]"
exit
fi
if [ -f $1 ]
then
dir="."
if [ $# -eq 2 ]
then
dir=$2
fi
inode=`ls -i $1|cut -d " " -f 2`
```

```
echo "Hard links of $1 are"
find $dir -inum $inode -print
echo "Soft links of $1 are"
find $dir -lname $1 -print
else
echo "The file $1 does not exist"
fi
```

Run1:

```
[root@localhost shellprgms]$ sh 5a.sh hai.c
Hard links of hai.c are
./hai.c
Soft links of hai.c are
./hai_soft
```

Program 9) This shell script displays the calendar for current month with current date replaced by * or ** depending on whether date has one digit or two digits.

```
#!/bin/bash
n=` date +%d`
echo " Today's date is : `date +%d%h%y` ";
cal > calfile
if [ $n -gt 9 ]
then
sed "s/$n/\**/g" calfile
else
sed "s/$n/\*/g" calfile
[root@localhost shellprgms]# sh 6a.sh
Today's date is : 10 May 05
May 2005
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6 7
8 9 ** 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

Program 10) This shell script implements terminal locking. Prompt the user for a password after accepting, prompt for confirmation, if match occurs it must lock and ask for password, if it matches terminal must be unlocked

```
trap “ " 1 2 3 5 20
clear
echo -e “\nenter password to lock terminal:"
stty -echo
read keynew
stty echo
echo -e “\nconfirm password:"
stty -echo
read keyold
stty echo
if [ $keyold = $keynew ]
then
echo "terminal locked!"
while [ 1 ]
do
echo "retype the password to unlock:"
stty -echo
read key
if [ $key = $keynew ]
then
stty echo
echo "terminal unlocked!"
stty sane
exit
fi
echo "invalid password!"
done
else
echo " passwords do not match!"
fi
stty sane
```

Run1:

```
[root@localhost shellprgms]# sh 13.sh
enter password:
confirm password:
terminal locked!
```

retype the password to unlock:
invalid password!
retype the password to unlock:
terminal unlocked!