Chapter 1: File attributes and permissions

1. The ls command with options.
2. Changing file permissions: the relative and absolute permissions changing methods.
3. Recursively changing file permissions.
4. Directory permissions.

1. The ls command with options.

ls –l :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**
  - The file type and its permissions are associated with each file.
- **Links**
  - Links indicate the number of file names maintained by the system. This does not mean that there are so many copies of the file.
- **Ownership**
  - File is created by the owner. The one who creates the file is the owner of that file.
- **Group ownership**
  - Every user is attached to a group owner. Every member of that group can access the file depending on the permission assigned.
- **File size**
  - File size in bytes is displayed. It is the number of character in the file rather than the actual size occupied on disk.
- **Last Modification date and time**
  - Last modification time is the next field. If you change only the permissions or ownership of the file, the modification time remains unchanged. If at least one character is added or removed from the file then this field will be updated.
- **File name**
  - 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-- 2 kumar metal 19555 may 10 15:45 chap02
drwxr-xr-x 2 kumar metal 512 may 09 12:55 helpdir
drwxr-xr-x 3 kumar metal 512 may 09 11:05 progs
```

The –d option: Listing Directory Attributes

$ls -d
This command 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 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.

2. Changing file permissions: the relative and absolute permissions changing methods.

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
```

rwx	**r-x**	**r- -**
owner/user	group owner	others

- The first group(rwx) has all three permissions. The file is readable, writable and executable by the owner of the file.
- The second group(r-x) has a hyphen in the middle slot, which indicates the absence of write permission by the group owner of the file.
- The third group(r- -) 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.

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: u – user g – group o – others a - all (ugo)**
- **Operations : + assign - remove = absolute**
- **Permissions: r – read w – write x - execute**

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
  $ls –l 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

```
$ls –l xstart
-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	-wx	write and execute
4	r--	read only
5	r-x	read and execute
6	rw-	read and write
7	Rwx	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
-rw-rw-rw- 1 kumar   metal   1906  sep 10  23:38 xstart

$chmod 644 xstart
-rw-r—r-- 1 kumar   metal   1906  sep 10  23:38 xstart


$chmod 761 xstart

-rwxrw--x 1 kumar   metal   1906  sep 10  23:38 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- - 1 kumar    metal    1906  sep 10  23:38 xstart
```

$chmod u-rw, go-r xstart or
$chmod 000 xstart

```
---------  1 kumar    metal    1906  sep 10  23:38 xstart
```

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

$chmod a+rwx xstart or chmod 777 xstart

```
-rwxrwxrwx 1 kumar    metal    1906  sep 10  23:38 xstart
```

- 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 .

3. <u>Using chmod Recursively</u>

$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.

4. <u>Directory Permissions</u>

- 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 super user 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 cannot 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 super user 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
```

Chapter 2: The Shells Interpretive Cycle

1. Wild cards.
2. Removing the special meanings of wild cards.
3. Three standard files and redirection.
4. Connecting commands: Pipe.
5. Basic and Extended regular expressions.
6. The grep, egrep.
7. Typical examples involving different regular expressions.

THE SHELL

- Shell acts as both a command interpreter as well as a programming facility.

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.
- 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.
- 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 meta characters 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.

1. Wild-Cards

- A pattern is framed using ordinary characters and a meta character (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 meta characters 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 z
[!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 z (Not in C Shell)
{pat1,pat2...}	Pat1, pat2, etc. (Not in Bourne shell)

The * and ?

- The metacharacter *, is one of the characters of the shell's wild card set.
- It matches any number of characters (including none).
- To list all files that begin with **chap**.
  ```
  $ ls chap*
  chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy chapz
  ```
- chap* matches the string chap. When the shell encounters this command line, it identifies the * immediately as a wild card.
- It then looks in the current directory and recreates the command line as below from the filenames that match the pattern chap*:
  ```
  ls chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy chapz
  ```

- ? matches a single character.
- To list all files whose filenames are six character long and start with chap.
  ```
  $ ls chap?
  chapx chapy chapz
  $ ls chap??
  chap01  chap02  chap03   chap04   chap15    chap16    chap17
  ```

Matching the Dot

- Both * and ? operate with some restrictions. for example, the * doesn't match all files beginning with a . (dot) or 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 .???*
  .bash_profile   .exrc   .netscape     .profile
  ```
- 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

- 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]
chap01   chap02    chap04

$ls chap[x-z]
chapx    chapy      chapz
```

<u>Negating the character class(!)</u>

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

***.[!co] -** To match all filenames with a single-character extension but not the .c or .o files

[!a-zA-Z]* - To match all filenames that don't begin with an alphabetic character

2. <u>Removing the special meanings of wild cards.</u>

Escaping and Quoting

- 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:

$ echo '\\'	*Displays a \*
$ rm 'chap*'	*Removes file chap**
$ rm 'My Document.doc'	*Removes file My Document.doc*

3. <u>Three standard files and redirection.</u>

- 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.

<u>Standard input</u>

The standard input can represent three input sources:

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

<u>How input redirection works:</u>

$ wc < sample.txt

1. On seeing the <, the shell opens the disk file, sample.txt for reading
2. It unplugs the standard input file from its default source and assigns it to sample.txt
3. wc reads from standard input which has earlier been reassigned by the shell to sample.txt

<u>Standard output</u>

The standard output can represent three possible destinations:

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

<u>How output redirection works:</u>

$ wc sample.txt > newfile

1. On seeing the >, the shell opens the disk file, newfile for writing
2. It unplugs the standard output file from its default destination and assigns it to newfile.
3. wc writes to standard output which has earlier been reassigned by the shell to newfile

<u>Standard error:</u>

- 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.

4. <u>Connecting commands: Pipe</u>

- With piping, the output of a command can be used as input (piped) to a subsequent command.
  Syntax:
  $ 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
```

- To count the number of lines and redirect wc's input so that the filename doesn't appear in the output:

```
$wc –l > user.txt
5
```

- Using an intermediate file(user.txt), we counted the number of lines.

- Method of running two commands seperately has 2 disadvantages:

1. For long running commands, this process is slow. The second command cant act unless the first has completed its job.
2. You need an intermediate file that has to be removed after completion of the job. When handling large files, temporary files can build up easily and eat up disk space in no time.

- The shell can connect these streams using a special operator, the | (pipe) and avoid creation of the disk file.
- The pipe is the third source and destination of standard input and standard output

Examples

$ ls -l | wc –l	**Displays number of file in current directory**
$ who | wc –l	**Displays number of currently logged in users**

- In a pipeline, all programs run simultaneously. A pipe has a built in mechanism to control the flow of the stream.
- Pipe is both being read and written, the reader and writer have to act in unison.
- If one operates faster than the other, then the appropriate driver has to readjust the flow.

5. The grep, egrep.

grep: Searching for a pattern

- grep scans its input for a pattern displays lines containing the pattern, the line numbers or filenames where the pattern occurs. The command uses the following syntax:

  $grep options pattern filename(s)

- grep searches for pattern in one or more filename(s), or the standard input if no filename is specified.

- The first argument (except the options) is the pattern and the remaining arguments are filenames.

Examples:

```
$ grep "sales" emp.lst

2233|a. k. Shukla      |g. m.     |sales |12/12/52|6000
1006|chanchal singhvi |director  |sales |03/09/38|6700
1265|s. n. dasgupta    |manager |sales |12/09/63|5600
2476|anil Aggarwal     |manager |sales |01/05/59|5000
```

here, grep displays 4 lines containing pattern as "sales" from the file emp.lst.

- grep silently returns the prompt because no pattern as "president" found in file emp.lst.

```
$ grep president emp.lst          #No quoting necessary here
$ _                               #No pattern (president) found
```

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

$ grep “director” emp1.lst emp2.lst

```
emp1.lst: 9876|jai sharma         |director |production
|12/03/50|7000
emp1.lst: 2365|barun sengupta    |director |personnel
|11/05/47|7800 emp1.lst: 1006|chanchal singhvi |director |sales
|03/09/38|6700
emp2.lst: 6521|lalit chowdury     |director |marketing
|26/09/45|8200
```

Here, first column shows file name.

grep options:

The below table shows all the options used by grep.

Option	**Significance**
-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	Matches multiple patterns
-f filename	Takes patterns from file, one per line
-E	Treats patterns as an ERE
-F	Matches multiple fixed strings

- **Ignoring case (-i):**

When you look for a name but are not sure of the case, use the -i (ignore) option.

$ grep -i 'agarwal' emp.lst

```
3564|sudhir Agarwal |executive |personnel |06/07/47|7500
```

This locates the name Agarwal using the pattern agarwal.

- **Deleting Lines (-v):**

The -v option selects all the lines except those containing the pattern.

It can play an inverse role by selecting lines that does not containing the pattern.

$ grep -v 'director' empl.lst

```
2233|a. k. shukla         |g. m.     |sales      |12/12/52|6000
5678|sumit chakrobarty    |d. g. m.  |marketing  |19/04/43|6000
5423|n. k. gupta          |chairman |admin      |30/08/56|5400
6213|karuna ganguly       |g. m.     |accounts   |05/06/62|6300
1265|s. n. dasgupta       |manager  |sales      |12/09/63|5600
```

```
4290|jayant choudhury     |executive |production |07/09/50|6000
2476|anil Aggarwal        |manager  |sales          |01/05/59|5000
3212|shyam saksena        | d. g. m. |accounts |12/12/55|6000
3564|sudhir Agarwal       |executive |personnel |06/07/47|7500
2345|j. b. saxena         |g. m. |marketing |12/03/45|8000
0110|v. k. Agrawal        |g. m. |marketing |31/12/40|9000
```

- **Displaying Line Numbers (-n):**

The -n(number) option displays the line numbers containing the pattern, along with the lines.

```
$ grep -n 'marketing' emp.lst

3: 5678  |sumit chakrobarty |d. g. m.  |marketing |19/04/43|6000
11: 6521|lalit chowdury      |director  |marketing |26/09/45|8200
14: 2345|j. b. saxena        |g. m.      |marketing |12/03/45|8000
15: 0110|v. k. Agrawal       |g. m.      |marketing |31/12/40|9000
```

here, first column displays the line number in emp.lst where pattern is found

- **Counting lines containing Pattern (-c):**

How many directors are there in the file emp.lst?
The -c(count) option counts the number of lines containing the pattern.

```
$ grep -c 'director' emp.lst
4
```

- **Matching Multiple Patterns (-e):**

With the -e option, you can match the three agarwals by using the grep like this:

```
$ grep -e "Agarwal" -e "aggarwal" -e "agrawal" emp.lst
2476|anil aggarwal |manager |sales |01/05/59|5000
3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. agrawal |g. m. |marketing |31/12/40|9000
```

- **Taking patterns from a file (-f):**

You can place all the patterns in a separate file, one pattern per line.
Grep uses -f option to take patterns from a file:

```
$ grep -f patterns.lst emp.lst

9876|jai sharma         |director    |production   |12/03/50|7000
2365|barun sengupta  |director    |personnel     |11/05/47|7800
5423|n. k. gupta         |chairman |admin          |30/08/56|5400
1006|chanchal singhvi |director   |sales           |03/09/38|6700
1265|s. n. dasgupta    |manager   |sales           |12/09/63|5600
2476|anil aggarwal     |manager   |sales           |01/05/59|5000
 6521|lalit chowdury  |director    |marketing    |26/09/45|8200
```

Basic Regular Expression (Bre)

- Like the shell's wild-cards which matches similar filenames with a single expression, grep uses an expression of a different type to match a group of similar patterns.
- Unlike shell's wild-cards, grep uses following set of meta-characters to design an expression that matches different patterns.
- If an expression uses any of these meta-characters, it is termed as **Regular Expression (RE).**

The below table shows the BASIC REGULAR EXPRESSION (BRE) character set-

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

- A RE lets you specify a group of characters enclosed within a pair of rectangular brackets, [ ], in which case the match is performed for a single character in the group.

```
$ grep '[aA]g[ar][ar]wal' emp.lst

3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. Agrawal  |g. m.     |marketing |31/12/40|9000
```

The *

- The * (asterisk) refers to the immediately preceding character.
- Here, it indicates that the previous character can occur many times, or not at all.

```
$ grep '[aA]gg*[ar][ar]wal' emp.lst

2476|anil Aggarwal  |manager  |sales      |01/05/59|5000
3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. Agrawal  |g. m.     |marketing |31/12/40|9000
```

The Dot

- A . matches a single character.
- The pattern 2... matches a four-character patten beginning with a 2.
- The pattern .* matches any number of characters, or none.

$ grep 'j.*saxena' emp.lst

```
2345|j. b. saxena |g. m. |marketing |12/03/45|8000
```

Specifying pattern locations (^ and $)

^ (carat) – For matching at the beginning of a line

$ (dollar) – For matching at the end of a line

$ grep '^2' emp.lst

```
2233|a. k. shukla      |g. m.     |sales       |12/12/52|6000
2365|barun sengupta |director  |personnel  |11/05/47|7800
2476|anil Aggarwal   |manager |sales        |01/05/59|5000
2345|j. b. saxena      |g. m.      |marketing |12/03/45|8000
```

$ grep '7...$' emp.lst

```
9876|jai sharma        |director    |production |12/03/50|7000
2365|barun sengupta |director    |personnel   |11/05/47|7800
3564|sudhir Agarwal |executive |personnel   |06/07/47|7500
```

Extended Regular Expression (Ere) and Egrep

- **ERE** make it possible to match dissimilar patterns with a single expression.
- **grep** uses ERE characters with -E option.
- **egrep** is another alternative to use all the ERE characters without -E option. This **ERE** uses some additional characters set shown in below table-

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

The + and ?

+ - Matches one or more occurrences of the previous character

? - Matches zero or one occurrence of the previous character.

$ grep -E “[aA]gg?arwal” emp.lst

```
2476|anil aggarwal |manager |sales |01/05/59|5000
```

```
3564|sudhir Agarwal |executive |personnel |06/07/47|7500
```

Matching Multiple Patterns(|, (and))

```
$ grep -E 'sengupta|dasgupta' emp.lst
```

```
2365|barun sengupta |director |personnel |11/05/47|7800
1265|s. n. dasgupta |manager |sales |12/09/63|5600
```

```
$ grep -E '(sen|das)gupta' emp.lst
```

```
2365|barun sengupta |director |personnel |11/05/47|7800
1265|s. n. dasgupta |manager |sales |12/09/63|5600
```