Module 2

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 read only 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.

By : Dr. Sudhamani M J
Associate Professor, Dept of CSE
RNSIT

File attributes and permissions

Chmod: Changing file permissions

chmod command is used to set the permission of one or more files for all 3 categories of users i.e user, group and others.

chmod command can be executed only by the user/owner and the superuser.

chmod 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

Changing permissions in a relative manner, chmod only *changes the permissions specified in the command line* and *leaves the other permissions unchanged*.

Syntax: chmod *category operation permission filenames(s)*

File attributes and permissions

Relative Permissions

chmod takes as its argument an expression comprising letters and symbols that completely describe the user category and the type of permission begin assigned or removed.

User *category* (user, group, others)
The *operation* to be performed (assign or remove a permission)
The type of *permission* (read, write, execute)

Category	Operation	Permission
u - User g - Group o - Others a - All	+ Assigns permission - Removes permission = Assigns absolute permission	r - Read permission w - Write permission x - Execute permission

File attributes and permissions

Relative Permissions

$ chmod u+x test

$ ls -l test
-rwxr--r-- 1 kumar …. test

If test is an executable file it can be executed by the owner.

To enable others to execute the file

$ chmod ugo+x test ; ls -l test

String ugo combines all three categories - user, group and others. Synonym for ‘ugo’ is ‘a’.
Other way round if no string is specified then permission is applied for all categories.

$ chmod +x test ; ls -l test

File attributes and permissions

Relative Permissions

chmod accepts multiple filenames.

$ chmod +x test a.c b.cpp

Permissions are removed with - operator. To remove read permission from both group and others

$ chmod go-r test ; ls -l test

chmod accepts multiple expressions delimited by commas.

$ chmod a-x,go+r test ; ls -l test

More than one permission can also be set for a file using chmod

$ chmod u+rwx test ; ls -l test

File attributes and permissions

Absolute Permissions

In absolute mode, permission is set through octal numbers.

Octal numbers use the base 8, and octal digits have the values 0 to 7, which means that a set of 3 bits can represent one octal digit.

Binary	***Octal***	***Permissions***	***Significance***
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and Executable

Binary	Octal	Permission	Significance
000	0	---	No permission
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable
100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwx	Readable, writable and executable

File attributes and permissions

Absolute Permissions

There are 3 categories and 3 permissions for each category, hence 3 octal digits can describe a file’s permissions completely.

Most Significant Digit represents user and the least one represents others.

$ chmod a+rw test

$ chmod 666 test ; ls -l test

-rw-rw-rw- …… test

$ chmod 761 test

File attributes and permissions

Recursively changing file permissions

It is possible to make chmod descend a directory hierarchy and apply the expression to every file and subdirectory, which is done by -R (recursive) option

$ chmod -R a+x dir

$ chmod 000 x.c

virtually useless

$ chmod 777 x.c

This type of command provides universal write permission.

File attributes and permissions

Directory permissions

Directories also have their own permissions and the significance of these permissions differ from ordinary file.

$ mkdir dir ; ls -ld dir d option to display only directory
drwxr-xr-x 2 kumar ….. dir

A directory must never be writable by group and others.

The Shell's Interpretive Cycle

Shell is the interface between user and the UNIX system.

Shell is a combination of command interpreter and a programming language.

Shell is also a process that creates an environment for user to work in.

When user key in a command, it will be considered as input to the shell. Shell first scans the command line for **metacharacters**(>, |, * etc).

Shell performs all actions represented by the symbol before the command can be executed.

$ rm -R *

* metacharacter is not part of rm command, but shell replaces * with all file names in the pwd. rm finally runs with these file names as arguments.

The Shells Interpretive Cycle

When all pre-processing is complete, shell passes on the command line to the kernel for final execution.

Command passed to kernel will not have any metacharacters that were originally used.

shell has to wait for notice of its termination from the kernel.

After the command has completed its execution, shell once again issues the prompt to take up next command.

Summary of shell in it interpretive cycle

1. Shell issues prompt and waits for user to enter command

2. After a command is entered, shell scans the command line for metacharacters and expands abbreviations to recreate a simplified command line.

3. Shell then passes on the command line to the kernel for execution

4. Shell waits for the command to complete.

5. After command execution is complete, prompt reappears and the shell returns to its waiting state to start the next cycle, where in next command can be entered.

Wild cards
$ ls chap*

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 the 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)

Wild cards The * and ?

The metacharacter * matches any number of characters, including none.

chap*

chap

chap01

chapx

chap*

$ echo *

$ rm *

Wild cards The * and ?
? wild-card matches a single character.

chap?

chapx

chapy

chapz

chap??
chap01 chap02 chap03….

Matching with Dot

$ ls .???* lists all hidden file names having at least 3 characters after the dot.

Wild cards Matching with Dot

Two important points related to * and ?

First, they will match a filename beginning with a dot, but they can match any number of embedded dots.

Ex: the pattern apache*gz matches

apache_1.3.20.tar.gz

Second * and ? don’t match the / in a pathname.

Ex: cd /usr?local

cannot be used to match cd /usr/local

The Character class

Compared to * and ? more restrictive patterns can be generated with **character class**.

rectangular brackets [and], but it matches a single character in the class.

Wild cards The Character class

[abcd] matches a single character

a

b

c

This can be combined with any string or another wild-card expression.

$ ls chap0[124]

chap01

chap02

chap03

Range specification is also possible inside the class with a - (hyphen); the two characters on either side of it form the range of characters to be matched.

$ ls chap0[1-4]
chap01
chap02
chap03
chap04

$ ls chap[x-z]

chapx
chapy
chapz

A valid range specification requires that the character on the left have a lower ASCII value, than the one on the right.

Note: Expression [a-zA-Z] matches all file names beginning with an alphabet, irrespective of case*

Wild cards The Character class

Negating the Character Class (!)

$ ls *.[!co]
not .c or
.o files.

[!a-zA-Z]*

matches all filenames that do not begin with an alphabetic character

Matching Totally Dissimilar Patterns

To list all C and java source programs in pwd
$ ls *.{c,java}

Wild cards

ls *.c

Lists all files with extension .c

mv * ../bin

Moves all files to bin subdirectory of parent directory

cp foo foo*

Copies foo to foo* (* loses meaning here).

cp ?????? progs

copies to progs directory all files with six character names.

lp note[0-1][0-9]
note00
note01

rm *.[!l][!o][!g]

Removes all files with three-character extensions except with

.log extension

cp -r /home/kumar/{include, lib, bin}

copies recursively the three directories, include, lib and bin from /home/kumar to the current directory.

Escaping and Quoting

File names in unix can be framed using metacharacters too (i.e *, ? etc.,)

Actual meaning of wild cards can be invalidated and can be considered as normal characters using two ways.

- **Escaping** - providing a \ before wild-card removes its special meaning.

- **Quoting** - Enclosing wild-card, or even the entire pattern, within quotes like '*'. Anything within these quotes are left alone by the shell and are not interpreted.

Escaping

```
$ cp > t\*
aaaa^D

$ ls  -l  t\*

$ echo > chap0\[1-3\]
```

Escaping and Quoting

Escaping the space

Apart from metacharacters, there are other characters that are special, like the *space character*.

The shell uses it to delimit command line arguments. A filename or directory name can be framed by two words, having a space in between them.

```
$ cat > My\ Doc ;  ls  My\ Doc
My Doc
$ rm My\ Doc
```

Escaping the \ Itself

In order to interpret \ itself literally, another \ must be appended before it.

```
$ echo \\
\
$ echo The newline character is \\n
The newline character is \n
```

Escaping and Quoting

Escaping the Newline Character

The newline character is also special, which marks the end of the command line.

To ensure better readability, a single line can be split into two lines by keying in \ before enter and the command can be continued in the second line.

```
$ ls \
> My\ Doc
My Doc
```

Escaping and Quoting
Quoting

$ echo ‘\’

$ rm ‘chap*’

$ rm “My Document.doc”

Using escaping character ‘\’ becomes tedious when there are many wild cards to be nullified. Quoting is often a better solution.

$ echo ‘The characters |, <, > and $ are also special’
Or
$ echo The characters \|, \<, \> and \$ are also special

Escaping and Quoting

Quoting

Double quotes are more permissive/liberal, they do not protect $ and the ` (backquote or backtick).

$ cat > ms.sh
#!/bin/bash
testing = `date`
echo “The date and time are: $testing”

$ chmod u+x ms.sh

$ ./ms.sh

Escaping and Quoting

Quoting

$ echo “Command substitution used `` while SHELL is evaluated using $SHELL”

Command substitution used while SHELL is evaluated using /bin/bash

`` is evaluated to null command

$ echo “List of files in pwd are `ls`, and shell used is $SHELL”

List of files in pwd are

X.c

Y.c

T.c

…. /bin/bash

$ echo ‘yu file is created `cat > yu` and path is $PATH’

Escaping and Quoting
Quoting
It is often crucial to select the right type of quote. Single quote protect all special characters.

Double quotes, lets a pair of backquotes be interpreted as *command substitution* characters, and the $ as a variable prefix.

There is also a reciprocal relationship between the two types of quotes; double quotes protect single quotes and single quotes protect double quotes.

$ echo “ ‘inside single quote’ “
‘inside single quote’

$ echo ‘ “inside double quote” ‘
“inside double quote”

Three standard files and redirection

“Terminal” is a generic name that represents the screen or keyboard.

Ex: Command output and error messages are displayed on the terminal i.e screen, users provide command input through the terminal i.e keyboard.

Shell associates 3 files with terminal - two for display and one for keyboard.

These special files are actually **streams** of characters which many commands see as input and output. A stream is simply a sequence of bytes.

When a user logs in, the shell makes available 3 files representing 3 streams. Each stream is associated with a default device.

Three standard files and redirection

- *Standard Input* - File or stream representing input, which is connected to keyboard.
- *Standard Output* - File or stream representing output, which is connect to display.
- *Standard Error* - File or stream representing error messages that arise from the command

Three standard files and redirection

Standard Input

Normal input for commands cat and wc are disk files.

When no disk file names are passed as an argument to these commands, they read the file representing **standard input.**

Standard input file represents three input sources

- ➢ Keyboard, the default source
- ➢ A file using *redirection* with < symbol (a metacharacter)
- ➢ Another program using a pipeline

Keyboard, the default source

$ wc
Standard input
Can be redirected *[Ctrl-d] 1 5 32*

Three standard files and redirection
Standard Input
wc read a file i.e the standard input file.

A file using redirection with the < symbol

It can **redirect** the standard input to originate from a file on disk. This reassignment or redirection requires the < symbol.

```
$ wc  <  x.c
3    14    71
```

Three standard files and redirection

Standard Input

Taking Input Both from File and Standard Input

A Command can take input from a file and standard input both, by using - symbol .

$ cat - foo

$ cat foo - bar

Three standard files and redirection

Standard Output

Commands displaying output on the terminal, writes to the **standard output** file as a stream of characters, and **not directly** to the terminal.

There are 3 possible destinations of this stream.

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

$ ls -l displays file names with long listing option on the output terminal screen

$ wc sample.txt > newfile

1. Upon encountering >, shell opens the disk file, newfile, for writing.
2. Shell unplugs the standard output file from its default destination and assigns it to newfile.
3. wc (and not the shell) opens the file sample.txt for reading
4. wc writes to standard output which has been earlier reassigned by the shell to newfile.

If ‘newfile’ does not exist it will be created first, if exists old contents of ‘newfile’ will be overwritten.

Three standard files and redirection

Standard Output

$ wc sample.txt >> newfile

$ cat *.c > progs_all.txt copies all contents of .c file in pwd to the file progs_all.txt

Contents of the file copied to progs_all.txt will not be known the above command can be improvised as below.

$ (ls -x *.c ; echo ; cat *.c) > progs_all.txt

echo command, inserts a blank line between the multicolumn file list and code listings.

A number, called a file descriptor, is associated with each of these files, as follows:

File descriptor 0	Standard input
File descriptor 1	Standard output
File descriptor 2	Standard error (diagnostic) output

Three standard files and redirection

Standard Error

Kernel maintains a table of file descriptors for every process running in the system. First three slots are generally allocated to the three standard streams in this manner.

- 0-Standard input
- 1-Standard output
- 2-Standard error

> and 1> mean the same thing.

```
$ wc   x.cpp   1>   newfile
```

< and 0< mean the same thing.

```
$   wc   0<      x.cpp
```

Three standard files and redirection

Standard Error

$ cat x.c considering x.c file is not available

cat: cannot open a file

$ cat x.c 2> err_file

$ cat > x.txt > x1.txt 2> x2.txt

$ cat x3.txt > x1.txt 2> x2.txt

Assuming x3.txt file does not exist, error message will be copied to x2.txt, instead of displaying error message on standard display screen.

Three standard files and redirection
Standard Error
HOW REDIRECTION WORKS
Command like ls writes to file descriptor 1, which can be verified when the command is executed.

To save ls output in foo, the shell has to manipulate this file descriptor before running ls. It closes standard output and then opens foo. **Since, the kernel allocates the lowest unallocated integer in the file descriptor table, foo is assigned the value 1.** ls output is thus captured in foo.

This implementation requires two processes. **Shell creates a copy of its own process, performs the descriptor manipulation in the copied process and even executes ls command. Shell's own file descriptors are then left undisturbed(in parent process).**

After completion of ls command the prompt will be displayed again, because the file descriptor 1 is already assigned to standard output file

Three standard files and redirection

Filters: *Using Both Standard Input and Standard Output*

Unix commands can be grouped into 4 categories based on the usage of standard input and standard output.

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 from standard input, but they write to standard output.

3. Command like **lp** that read standard input but will not write to standard output.

4. Commands like **cat, wc, od, cmp, gzip,** etc., that use both standard input and standard Output.

Commands in fourth category are called, in Unix parlance, **filters.** The dual stream handling feature makes filters powerful text manipulators. Most filters can also read directly from files whose names are provided as arguments.

Three standard files and redirection

Filters: *Using Both Standard Input and Standard Output*

$ cat > calc.txt
2^32
25*50
30*25 + 15^2
^D

bc is a command in unix which does the job of calculator.

$ bc < calc.txt > result.txt

$ cat result.txt

Connecting Commands

Pipe

Pipe in Unix is used to pass the output of one process/command to another process/command.

$ who > user.txt

Further, wc command can be used to count number of lines in user.txt, which is related to the number of users, using -l option

$ wc -l user.txt

Two commands are executed separately, which has two obvious disadvantages.

1. For long running commands, this process can be slow. Second command cannot start execution unless the first has completed its job.

2. An intermediate file has to be removed after completion of the job. When dealing with huge amount of data, files holding on to huge amount of data has to be created, which consumes more disk space.

Connecting Commands

Pipe

$ who | wc -l

Shell can connect streams with a special operator, the | (pipe), and avoid creation of the disk file.

Output of who is piped to wc. When multiple commands are connected this way a **pipeline** is said to be formed. It is the shell that sets up this connection and the commands have no knowledge of it.

Pipe is the third source and destination of standard input and standard output respectively.

$ ls | wc -l counts the number of files in pwd, ls lists the files in pwd, one file in one line and wc counts the number of lines.

$ ls | wc -l > file_name.txt

Connecting Commands

Pipe

There is no restriction on the number of commands that can be used in pipeline.

$ cat > x.txt | wc | cat > result.txt

x.txt contents will passed as an input to wc command and output of wc command is stored in result.txt

Most of the times in a pipeline all programs are executed simultaneously. A pipe also has a built-in mechanism to control the flow of the stream.

Since, pipe is both read and written, the reader and writer has to act in unison. If one operator is faster than the other, then the appropriate driver has to readjust the flow.

Connecting Commands
Basic and Extended Regular Expression
grep: Searching for a pattern

Unix has a special group of commands for handling search requirements. grep command is one of them.

grep scans its input pattern in a file and displays lines containing the pattern, the line number or filenames where the pattern occurs.

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.

$grep “include” *

Since, grep is a filter, it can search its standard input for the pattern and also save the standard output in a file

grep: Searching for a pattern

It is not necessary to enclose pattern in quotes (‘ ‘ or “ “), if more than one word is to be searched using grep command, then it has to be compulsorily enclosed in quotes.

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

$ grep “int main” x.c x2.c x3.c

grep Options

Option	Significance
-i	Ignores case for matching
-v	Does not display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only

Option	Significance
-e exp	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen.
-x	Matches pattern with entire line
-f file	Takes patterns from file, one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings

Connecting Commands

grep Options

```
$ cat > emp.lst
1006|chanchal singhvi  |director |sales |03/09/38|6700
6521|lalit chowdary     |director |marketing |26/09/45|8200
9876|jai sharma         |director |production | 12/03/50|7000
3564|Sudhir Agarwal   |executive | personnel | 06/07/47| 7500
```

Ignoring Case (-i)

```
$grep -i ‘agarwal’ emp.lst
3564|Sudhir Agarwal   |executive | personnel | 06/07/47| 7500
```

Deleting Lines(-v)

```
$ grep -v ‘director’ emp.lst
3564|Sudhir Agarwal   |executive | personnel | 06/07/47| 7500
```

1006|chanchal singhvi |director |sales |03/09/38|6700
6521|lalit chowdary |director |marketing |26/09/45|8200
9876|jai sharma |director |production | 12/03/50|7000
3564|Sudhir Agarwal |executive | personnel | 06/07/47| 7500

Connecting Commands
grep Options
Displaying Line Numbers(-n)

$ grep -n ‘marketing’ emp.lst
2:6521|lalit chowdary |director |marketing |26/09/45|8200

Counting Lines Containing Pattern (-c)
$ grep -c ‘marketing’ emp.lst
1

$ grep -c ‘director’ emp*.lst
emp.lst:4
emp1.lst:2

Connecting Commands
grep Options
Displaying Filenames (-l)

```
1006|chanchal singhvi  |director |sales |03/09/38|6700
6521|lalit chowdary    |director |marketing |26/09/45|8200
9876|jai sharma        |director |production | 12/03/50|7000
3564|Sudhir Agarwal    |executive | personnel | 06/07/47| 7500
```

$ grep -l ‘manager’ *.lst
Desig.lst
emp.lst
emp1.lst

Matching Multiple Pattern (-e)
$ grep -e ‘dir’ -e ‘mark’ -e ‘direct’ emp.lst

Regular Expressions - An Introduction

It is tedious to specify each pattern separately with -e option.

Like shell’s wild cards which match similar filenames with a single expression, grep uses an expression of a different type to match a group of similar patterns.

Unlike, wild cards, this expression is a feature of the ***command*** that uses it and has nothing to do with the shell.

Command uses an elaborate metacharacter set, overshadowing the shell’s wild-cards and can perform amazing matches.

Regular Expressions - An Introduction

Symbols or Expression	***Matches***
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg, etc
.	A single character
.*	Nothing or any number of characters
[pqr]	A single character p, q or r
[c1-c2]	A single character within the ASCII range represented by c1 and c2
[1-3]	A digit between 1 and 3
[^pqr]	A single character which is not a p, q or r

Regular Expressions - An Introduction

Symbols or Expression	***Matches***
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern pat at beginning of line
pat$	Pattern pat at end of line
bash$	Bash at end of line
^bash$	Bash as the only word in line
^$	Lines containing nothing

Regular Expressions - An Introduction

If a regular expression in the command uses any of the characters, listed in the previous table, it is termed as **regular expression.**

Regular expressions take care of some common query and substitution requirements.

Ex: To list similar names, in which required one can be selected.

To replace multiple spaces with a single space

To display lines that begin with a #.

Two categories of regular expressions exist - *basic* and *extended.*

Basic Regular Expressions (BRE) ***Extended Regular Expression (ERE)***

grep command supports BRE by default and ERE with -E option.

The Character Class

A RE facilitates user to 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.

Regular Expressions
The Character Class
$ grep [ra] emp.lst   - matches either r or an a
hence matches Agarwal and Agrawal

The RE - [aA]g[ar][ar]wal
‘agarwal’
‘Agarawal’

here the model [ar][ar]   matches with any 4 pattern --aa ar   ra   rr

$ grep “[aA]g[ar][ar][wal]” emp.lst
3564|Sudhir Agarwal |executive | personnel | 06/07/47| 7500
1563|v. k agrawal |executive | personnel | 16/01/70| 3500

Regular Expressions

Negate Class (^)

RE use ^ to negate the character class, when a character class begins with this character, all characters other than the ones grouped in the class, are matched.

Ex: [^a-zA-Z] matches a single non-alphabetic character string.

The *

* (asterisk) refers to the *immediately preceding* character. In grep command it indicates that the preceding character can occur many times, or not at all.

g*

Matches single character g, or any number of g's. Since, there is a possibility that the previous character (i.e g in example) may not occur at all, it also matches a null string. Hence, apart from null string, it also matches following strings: g gg ggg gggg ……

"Zero or more occurences of the previous character"

In order to match a string beginning with g; gg* has to be used.

Regular Expressions

The *

$ grep ”[aA]gg*[ar][ar]wal” emp.lst

The pattern matches “aggarwal”, “Agarwal” and “agrawal” all three similar names but with different spellings. Pattern considered is not limited to match only three names.

The Dot

A ‘.’ matches a single character.

Ex: 2… matches four character pattern beginning with a 2.

RE .*, signifies any number of characters or none. In order to find a name that starts with ‘j’, the pattern will be

$ grep “j.*” emp.lst

RE to find “J.B. saxena” or “J.B. agarwal”

$ grep “J\.B\..*” emp.lst

Actual meaning of dot is escaped with \, which is used for despecializing next character.

Regular Expressions

Specifying Pattern Locations (^ and $)

Most of RE characters are used for matching patterns, but ^ and $ are used to match a pattern at the beginning or end of a line.

^ - For matching at the beginning of a line $ - For matching at the end of a line.

$ grep “^2” emp.lst to list those employees, whose employee number starts with 2.

$ grep “7…$” emp.lst

Lists those employees, whose salary lies between 7000 to 7999. ‘$’ will force grep command to search at the end.

$ grep “^[^2]” emp.lst Selects only those line whose emp-id’s do not begin with a 2.

$ ls -l | grep “^d” to list only directories in pwd

Regular Expressions

ERE and grep

ERE makes it possible to match dissimilar patterns with a single expression.

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
(lock/ver) wood	Matches lockwood or verwood

Regular Expressions

ERE and grep

ERE expressions are executed with -E option

The + and -

ERE set includes 2 special characters + and ?. They are often used in place of * to restrict the matching scope.

+	Matches one or more occurrences of the previous character
?	Matches zero or one occurrence of the previous character

Ex: b+ match b, bb, bbb etc., The expression b? Matches either a single instance of b or nothing.

$ grep -E "[aA]gg?arwal" emp.lst

This command is used to find employee names like *"anil **aggarwal**"* or *"sudhir **Agarwal**"*

Regular Expressions

The + and -

To list

#include <stdio.h>, #include <stdio.h>, #include <stdio.h>,

which is a multiword string. Considering there is no space between '#i' and there may be more than one space between 'include' and '<stdio.h>'.

$ grep -E "# ?include +<stdio.h>' emp.lst

Matching Multiple Patterns (|, (and))

The | is the delimiter of multiple patterns.

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

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

ERE when combined with BRE form very powerful RE. The expression 'agg?[ar]+wal' contains characters from both sets.

Shell Programming

Activities of shell are not restricted to command interpretation alone.

Shell with its whole set of internal commands can be considered as a language.

A shell program executes in *interpretive* mode.

Each statement will be loaded into memory when it is executed. Shell scripts consequently run slower than those written in high-level languages.

Shell programs are considered as powerful, because the external Unix commands blend easily with the shell's internal constructs.

Ordinary and Environment Variables

Variables play an important role in every programming language.

In Unix two types of variables are used 1. Ordinary 2. Environment variables

A variable in a shell script is a means of **referencing** a **numeric** or **character value**.

Unlike formal programming languages, a shell script doesn't require users to **declare a type** for variables.

Environment variables

These are the variables which are created and maintained by **Operating System itself.**

Generally these variables are defined in **CAPITAL LETTERS**.

Environment variables list can be seen using the command "**set**".

Environment variables

System defined variables	Meaning
BASH (/bin/bash)	Shell Name
BASH_VERSION (4.1.2(1))	Bash Version
COLUMNS (80)	No. of columns in screen
HOME (/home/dev)	Home Directory of User
LINES (25)	No. of columns in screen
LOGNAME (dev)	dev logging name
OSTYPE (linux-gnu)	OS type
PATH (/usr/bin:/sbin)	Path Settings
PWD, SHELL, USERNAME, PS1	

Introduction

When a group of commands have to be executed regularly, they will be stored in a file, and the file is executed as a **shell script** or **shell program**.

File which contains shell scripts can have .sh extension (not mandatory).

Shell scripts are executed in a separate child shell process. By default, the child and parent shell belong to the same type, but the first interpreter line of the script can be used to convey which shell to be used.

```
$ cat script.sh
#! /bin/sh
# script.sh:  Sample shell script
echo “Today’s date: `date`”
Echo “My shell: $SHELL” ^D
```

Introduction

Line which starts with # is a comment line except the first one which starts with #!, which requests the shell interpreter to choose the shell in which the following commands will be executed.

By, default execute option, will not be provided for the files that are created in unix. Files that contain shell scripts must have x option for execution, which will be done using chmod command.

$ chmod u+x script.sh

$ ./script.sh ‘ ./ ’ is conveying to the executor that the file is in pwd.

Explicitly execution of script in a different shell

$ ksh script.sh

First line in the script will be neglected when executed in this manner.

Shell Programming

Environment variables

Values of environment variables can be displayed using echo command

$ echo $OSTYPE

Ordinary variables

These variables are defined by **users**.

Variables allows to **store data temporarily** and use it throughout the script.

User variables can be any text string of up to **20 letters**, **digits**, or **an underscore character**.

User variables are case sensitive, so the variable Var1 is different from the variable var1.

Values are assigned to user variables using an **equal sign.** No spaces can appear between the variable, the equal sign, and the value.

var1=10 var2=-57 var3=testing var4="still more testing"

Shell Programming

Ordinary variables

The shell script **automatically determines data type** of the variable.

Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes its execution.

```
$ cat test3.sh
#!/bin/bash
# testing variables
days=10
guest=”Katie”
echo “$guest checked in $days days ago”
days=5
guest=”Jessica”
echo “$guest checked in $days days ago”
```

$ chmod u+x test3.sh $ ./test3.sh

Ordinary variables

Each time the variable is **referenced**, it produces the value currently assigned to it.

When referencing a variable value **dollar sign** has to be used. When a value is assigned to a variable dollar sign must not be used.

The .profile file

In Unix, .profile files contains definitions for a shell environment, such as environment variables, scripts to execute, and other instructions; used for storing pre-defined settings that are loaded when a shell program starts.

When opening a terminal(bash shell) in Unix, the program automatically searches for a ".profile" / ".bash_profile" file and executes it line by line as a shell script.

To manually run a profile file, use the command **source ~/.profile.**

Shell Programming

The .profile file

PROFILE files are hidden files that do not have a filename prefix. They are always named .profile and are located in the user's home directory.

To manually run a profile file, use the command **source ~/.profile.**

The .profile file is present in home ($HOME) directory and can be customized for individual working environment.

read: Making Scripts Interactive

read statement is used to accept input from the user, making scripts interactive.

Inputs read from the standard input device is placed in a variable.
Ex: read name
Script will pause at that point to accept input from the keyboard. Entered value will be stored in name.

Shell Programming

read: Making Scripts Interactive

Ex: read name

Since accepting a value is a form of assignment, $ will not be used before name.

$ vi cfile.sh

```
#! /bin/sh
echo "Enter the word to be searched: \n"
read pname
echo "Enter the file to be used: \c"
read fn
echo "Searching for $pname from file $fn"
grep "$pname" $fn     # pattern has to passed in “ “ for grep, if more than 1 word is considered in pattern.
echo "Selected records are shown above"
```

Above script will search for the content entered in pname in the file name entered in fn.

Shell Programming

read: Making Scripts Interactive

Ex: read pname filename

Read statement can be used to read multiple values from the same statement.

If the number of arguments supplied is less than the number of variables accepting them, any leftover variables will simply remain unassigned.

When the number of arguments exceeds the number of variables, the remaining words are assigned to the last variable.

```
$ vi test.sh
#! /bin/sh
echo "Enter the word to be searched: \n"
read pname fname
echo $pname  $fname
```

Input: first pname=first fname=" " **Output:** first

Input: first second word pname=first fname=second word **Output:** first second word

Shell Programming

readonly variables

Shell provides a way to mark variables as read-only by using the **read-only** command.
After a variable is marked read-only, its value cannot be changed.

```
$ vi test.sh
#! /bin/sh
readonly  var=”abc”
echo $var
var = “iop”              # this generates while executing the script.
```

Command Line Arguments

Shell scripts accept argument from the command line. Passing information or input to script from command line is considered as noninteractive.

When arguments are specified with a shell script, they are assigned to certain special variables.

Shell Programming : **Command Line Arguments**

First argument passed from command line while executing shell script, will be read into the variable $1, second argument into $2. These are termed as positional parameters.

Shell Parameter	**Significance**
$1, $2, ….	Positional parameters representing command line arguments
$#	Number of arguments specified in the command line
$0	Name of executed command
$*	Complete set of positional parameters as a single string
$?	Exit status of the last command
$$	PID of the current shell
$!	PID of the last background
“$@”	Each quoted string treated as a separate argument (recommended over $*)

Shell Programming

Command Line Arguments

```
$ vi test.sh
#! /bin/sh
echo “Program:  $0
The number of arguments specified is $#
The arguments are $*”
grep  “$1”    $2
echo “\nJob over”

$ chmod u+x test.sh
$ ./test.sh   malloc     a.c
```

Shell Programming

exit and Exit status of a command

Shell script uses exit command to terminate the execution.

exit 0 used when no error occurred

exit 1 used when error occured

Ex: If a grep command could not locate a pattern, error message will be displayed by the grep command. grep code will invoke exit(1) command, where in value 1 is communicated to the calling program, usually the shell.

It's only through exit, that every command returns an **exit status** to the caller.

A command is said to return a **true** exit status if it executes successfully, and **false** if it fails.

$ cat foo

Cat: can't open foo

Returns a nonzero exit status because it could not open the file. The shell offers a variable ($?) and a **test** command that evaluates a command's exit status.

Shell Programming

exit and Exit status of a command

$ vi scripts.sh

#! /bin/sh

echo “first statement in script”

echo “second statement

$ ./scripts.sh

Error message will be displayed

$ echo $?

2

this is the exit status of scripts.sh file while executing, if the script file is successful in execution output from echo command will be 0

The parameter $? Parameter $? Stores the exit status of the last command. It has the value 0 if the command succeeds and a nonzero value if it fails. This parameter is set by **exit**’s argument.

Shell Programming

exit and Exit status of a command

The parameter $?

```
$ cat > file
director
^D
$grep  director  file > /dev/null;  echo $?
0
$grep  dir  file > /dev/null;  echo $?
1                                                    ;  Failure  in  finding
pattern
$grep  dir  file1 > /dev/null;  echo $?
2                                                    ; Failure in opening file
“file1”
```

Exit status is extremely important for a program, because the program logic, that branches into different paths are depended on the success or failure of a command.

Shell Programming

Logical operators && and || for Conditional Execution

Shell provides 2 operators that allow conditional execution of the script, && and ||.

Syntax: cmd1 && cmd2 cmd1 || cmd2

In && operator cmd2 will be executed only when cond1 succeeds.
In || operator cmd2 is executed only when the first fails.

```
$ grep "director" file && echo "Pattern found"
director
Pattern found

$ grep "direcr" file && echo "Pattern found" ;output will be empty

$ grep "direcor" file || echo "Pattern not found"
Pattern not found
```

Shell Programming

Logical operators && and || for Conditional Execution

```
$ vi   cfile.sh
#! /bin/sh
echo "Enter keyword to find"
read kw
echo "Enter filename to find in"
read fn
grep "$kw"  $fn  || exit 2
echo "Pattern found"
```

; **Input:** malloc a.c **Output:** Pattern found

```
$ vi  cfile.sh
#! /bin/sh
grep  “$1”    $2  || echo “Pattern not found”
exit 2
```

; value 2 if passed to another program can be used logically,

but shell just recieves and neglects the value.

Shell Programming

The if Conditional

The if statement makes two way decisions depending on the consideration of a condition.

Shell uses the following forms of if condition.

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 … else … Fi
Form 1	**Form 2**	**Form 3**

Shell Programming

The if Conditional

If evaluates the success or failure of the command that is specified in front of it.

If the command succeeds, the sequence of commands following if is executed.

If the command fails, then the statements in else part is executed.

Statements of else part is not compulsory.

Every if is closed with a corresponding fi, and error will be encountered if “fi” is not present.

```
$ vi  test.sh
#! /bin/sh
if    grep   “$1”    /etc/passwd   2>/dev/null
then
    echo “Pattern found - Job Over”
else
    echo “Pattern not found”
fi                                                                    $
 /test.sh    ftp              $ /test.sh    ftp1
```

Shell Programming

The if Conditional

$ ./test.sh ftp
ftp:*.325…….
Pattern found - Job over

$ ./test.sh abcd
Pattern not found

The test command and its shortcut

If command, in shell script cannot handle, the values generated by relational expressions.

test command will be used to convert the value generated by relational expression, in a manner that can be interpreted by ‘if’ command.

test command works in 3 ways:

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

Shell Programming

The test command and its shortcut

‘test’ command will not display any output, but simply sets the parameter $?

Numeric Comparison

Numerical comparison operators used by test has an - (hyphen), followed by a two-letter string, and enclosed on **either side by whitespace**.

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

Shell Programming

Numeric Comparison

Numeric comparison in the shell is confined to integer values only; decimal values are simply truncated.

```
$ x=5;  y=7;  z=7.2
$ test $x -eq $y  ;  echo $?
1
```

Not equal

```
$ test $x -lt $y  ;  echo $?
0
```

True

```
$ test $z -eq $y   ; echo $?
bash: test: 7.2: integer expression expected
2
```

Error

```
$ vi   test.sh
#! /bin/sh
if  test  $#  -eq  0
then
        echo "Usage: $0 pattern file"
elif  test  $#  -eq  2
then
        grep "$1"   $2   ||   echo "$1 not found
in $2"
else
        echo "Two arguments are required"
fi
```

```
$ ./test.sh  printf    add.c

$ ./test.sh     prif     add.c
prif not found in add.c
```

Shell Programming

Shorthand for test

test command is widely used, hence there is a shorthand method of executing it. A pair of rectangular brackets enclosing the expression can be used instead of test command.

Ex: test $x -eq $y

[$x -eq $y]

Whitespace is mandatory around the operator

String Comparison

test can be used to compare strings. 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

Test	**True if**
stg	String stg is assigned and not null
s1 == s2	String s1 == s2 (Korn and Bash only)