The SHELL'S INTERPRETIVE CYCLE

When you log onto a UNIX machine , you see a prompt. This prompt remains until you key in something. Even though it may appear that the system is idling, a UNIX command is in fact running at the terminal. But this command is special its with you all the time and never terminate unless you log out. This command is **shell**.
If you provide the input in the form of ps command (that shows processes owned by you), you will see shell running

```
$ps
PID    TTY    TIMECMD
328    pts/2  0:00   bash
```

- The bash shell is running at the terminal /pts/2.
- When you key in the command it goes to shell as input.
- The shell scans the command line for metacharacters.These are the characters that mean nothing to the command but has special meaning to theshell.
- For example if the shell encounters metacharacters like *,| etc in the commandline.
- If the metacharacter is *, then shell replaces it with all the filenames in the currentdirectory.
- When all preprocessing is complete , the shell passes on the command to the kernel for ultimate execution.
- While the command is running, the shell has to wait for notice of its termination from the kernel.
- After the command is complete with its execution, then shell once again issues the prompt to take up your nextcommand.

ACTIVITIES PERFORMED BY THE SHELL IN ITS INTERPRETIVE CYCLE.

1. The shell issues the prompt and waits for you to enter a command.
2. After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like * in rm *) to recreate the simplified command line.
3. It then passes on the command line to the kernel forexecution
4. The shell waits for the command to complete and normally cant do any work while the command is running.
5. After command execution is complete, the prompt re appers and shell returns to its waiting role to start the next cycle. You can now enter the next command.

2. WILD CARDS AND FILE NAMEGENERATION.

2.1 WILDCARDS:

The metacharacters taht are used to construct the generalized pattern for matching filenames belong to the category called **wild cards.**

2.1.1 The * and?

- **The metacharacter *** is one of the characters of the shell wild card set. It matches any number of characters including none.
- For example : tomatchfilenames chap chap01 chap02 chap03 chap04

$ ls chap*
Output : // the * matches all strings along with none

chap
chap0
1
chap0
2
chap0
3
chap0
4

- **The metacharacter ?**matches a singlecharacter
  For example: tomatchfilenames chapx chapy chapz
  $ ls chap?
  Output: //the ?replaces single character i,e ? is replaced by
  x,y,z chapx
  chap
  y
  chapz

2.1.2 Matching thedot(.)

- The .dot metachacter can be used to match all the hidden files in yourdirectory.
- Example: To list all hidden files in your directory having atleast three characters after thedot.
  $ ls .???*
  Output:
  .bash_profile
  .exrc
  .netscape
  .profile

2.1.3 The character class []

- This class comprises a set of charcters enclosed by the rectangular brackets [and],butit matches only a single character in theclass.
- The **pattern [abcd]** is a character class and it matches a single character a, b, c ord
- For example tomatchchap01 chap02 chap03 chap04
  $ ls chap0[1234]
  output:
  chap0
  1
  chap0
  2
  chap0
  3
  chap0
  4

2.1.4 Negating the characterclass(!)

- It is used to reverse the matchingcriteria.

- Forexample:
  To match all the filenames with single character extensions but not .c or .o files
  $ ls *.[!co]
  *** to match any filename**
  . extension
  !except
  [!co]---> .c extension or .o extension
- To match filenames that does not begin with adigit
  $ ls [!0-9]*

- To match filename with 3 character that does not begin with an Upper Caseletter
  $ ls [!A-Z]??

2.2 REMOVING THE SPECIAL MEANINGS OF WILDCARDS

ESCAPING and QUOTING

Escaping: providing a \(backslash character) before the wild card to remove or escape its special meaning.
Quoting: enclosing the wild card or even the entire pattern within quotes ('chap* '). Anyting within the quotes are left alone by the shell and not interpreted.

2.2.1 ESCAPING

- Placing a \ immediately before a metacharacter turns off its specialmeaning.
- For instance * , matches * itself. Its special meaning of matching zero or moreoccurrencesof character is turned off.
- Ex1:
  $rm chap*
  removes all the filenames starting with chap. Chap, chap01,chap02 and chap03 are removed.

  $rm chap* // * metacharacter meaning is turnedoff removes the filenamewithchap* /*name of the file itself ischap*.

- Ex2:
- If there are files with names chap01, chap02,chap03.
- To list the filenames starting withchap0
  $ ls chap0[1-3] Output:
  chap0
  1
  chap0
  2
  chap0
  3

  To match the file named as chap0[1-3]
  $ ls chap0\[1-3\] Output:
  chap0[1-3]

Escaping the space: To remove the file My document.doc, which has space embedded,
$rm My\ document.doc

Escaping the newline character.
$echo -e "The newline charcter is \n Enter the command"
Output:
Thenewlinecharacter /n – newline character is interpreted and cursor moves to nextline

```
Enter the command
$ echo "the newline character is \\n enter the command"
Output:
 the newline character is \\n enterthecommand    /* here newline character interpretationis
```

turned off and the \n is printed as it is*/

Escaping the \ itself
**$echo **
Output
****

2.2.2QUOTING:

- This is the another way of turning off the meaning ofmetacharacter.
- When a command argument is enclosed within quotes, the meaning of all enclosedspecial characters are turnedoff

$rm'chap*' // * metacharacter meaning is turnedoff
removes the filenamewithchap* /*name of the file itself ischap*.

$rm "My\document.doc" /* Toremove the file My document.doc, which has space embedded.

$echo '\'
output
****

3. REDIRECTION: THE THREE STANDARDFILES

Redirection is the process by which we specify that a file is to be used in place of one of the standard files.

- With input files, we call it input redirection;
- With output file, we call it output redirection
- With the error file, we call it error redirection.

Standard input: The file representing the input, which is connected to the keyboard Standard output: The file representing the output, which is connected to the display.
Standard error: the file representing the error messages that emanate from the command or shell. This is also connected to display.
Each of the three standard files are represented by a number called a **file descriptor.** The first three slots are generally allocated to three standard streams in this manner 0: standard input
1: standard output
2: standard error

3.1 STANDARD INPUT

This file is indeed special

- The keyboard, the default source
- **a file using redirection with the < symbol**
- another program using thepipeline

- The input redirection operator is less than character(<).
- When you use wc without an argument , it prompts you to provide the input from standardinput keyboard
  $ wc
  Unix is a multiuser multitasking
  OS [ctrl-d]
- When wc is used with argument. Filename is passed as an argument i,e wc takes the

input from the filename we havespecified

- For example: Create a file with namesample.txt
  $ vi sample.txt
  Unix is a multiuser multitasking OS

  :wq /*saving and quitting from thefile

  $ wc< sample.txt /*wc command takes input from the filesample.txt
  output
  1 6 36 /* count of characters, words, lines of filesample.txt

3.2 STANDARD OUTPUT

- All commands displaying the output on the terminal actually write to the standard output fileas a stream of characters and not directly to the terminal assuch.
- There are three possible destinations of thisstream
- The terminal, the defaultdestination
- **A file using the redirection symbol > and>>**
- As input to another program using apipeline
- There are two basic redirection operators for standardoutput.

  a. greater than character(>):

  If you want the file to contain only the output from this execution of the command, you can use greater than token.
  Ex: consider the file **sample.txt**
  $catsample.txt /* to display the content ofsample.txt
  Unix is a multiuser multitasking OS

  $ wc sample.txt > newfile
  >symbolredirects the output of wc command to a file named newfile The output is now stored in newfile

  $cat newfile /* To view the content ofnewfile

  1 6 37

  b Two greater than characters>> (append)
  If you want the output of the command to be appended without overwriting the existing content.

  $who >> newfile
  The output of who command is appended to newfile without overwriting the existing content.
  $cat newfile
  1 6 37 /* output of wc command executedpreviously

  root console aug 1 07:51 (:0) /* output of whocommand
  kumar pts/10 Aug 1 02:51 (:0)
  sharma pts/6 Aug 1 03:51 (:0)

3. 3.STANDARD ERROR

When you enter an incorrect command or try to open a non existent file, certain diagnostic messages show up on the screen. This is the standard error stream whose default destination

is the terminal.

Standard output and error on monitor
Ex 1: Consider two files file1 and file2 , where file1 exist and file2 do not exist
$ls –l file1file2
-rwxr— r-- 1 gilberg staff 1234 oct file1
Cannot access file2: no such fileordirectory /*error because file2 do notexist

Ex 2: To redirect standard output to same file
$ls -l file1 file2 1>filelist 2>filelist
The 1st argument is file1 and 2nd argument is file2. The output and errors are sent to file named filelist

$cat filelist
-rwxr— r-- 1 gilberg staff 1234 oct file1
Cannot access file2: no such file ordirectory

Ex 3: To redirect standard output to differentfiles
$ls -l file1 file2 1>stdout 2>stderr
The 1st argument is file1 and 2nd argument is file2.
The output of 1st file is sent to filename stdout and errors are sent to file named stderr.

$cat stdout
-rwxr— r-- 1 gilberg staff 1234 oct file1

$cat stderr
Cannot access file2: no such file or directory

4. CONNECTING COMMANDS: PIPE.

- We often need to use a series of commands to complete a task. For example, if we need to see list of users logged into the system, we use who command.However if we need a hard copy of the list, we need twocommands.
- First use **who** command to get the list and store the result in file usingredirection
- We then use **lpr** command to print thefile
- We can avoid the creation of intermediate file by using apipe

Pipe is an operator that temporarily saves the output of one command in a buffer that is being used at the same time as the input of the next command.

- The first command must be able to send its output to standard output. The second command must be able to read its input from standard input.
- **The token for a pipe is vertical bar(|)**
  $ who | lpr
  We use the **who command** because it reads from the system and sends the list of users to standard output.The pipe command uses a buffer to send the piped data to next command.The recieving command must recieve its data from standard input.
  $who
  root console aug 1 07:51 (:0)
  kumar pts/10 aug 1 02:51 (:0)
  sharma pts/6 aug 1 03:51 (:0)

```
rajath pts/8        aug    1      06:51 (:0)
```

vikas pts/14 aug 1 09:51(:0)

$who | wc-l
Output:5 /* count of number of lines of whocommand
Here the output of**who**command has been passed directly as the input to **wc** command and
whois said to be piped to **wc.**

5. SPLITTING THE OUTPUT: teeCOMMAND

- The tee command is an external command and handles a character stream by duplicating its input.
- The tee command copies standard output and at the same time copies it to one or morefiles.
- The first copy goes to standard output i,e monitor and at the same time the output is sent to the optional files specified in the argumentlist.
- The tee command creates the output files, if they do not exist and overwrites them, if they alreadyexist.

Ex:
The following command sequence uses **tee** to display the output of **who**and saves this output in a file as well.

$ who|tee user.txt

```
root     console   aug  1   07:51 (:0)
kumar    pts/10    aug  1   02:51 (:0)
sharmapts/6        aug  1   03:51 (:0)
```

The output of who will be displayed on the monitor and at the same time it will be saved in a file
user.txt

6.COMMANDSUBSTITUTION

- When a shell executes a command, the output is directed to standard output. Most of the time the standard output is associated with the monitor.
- There are times, however such as when we write complex commands or scripts that we need to change the output to a string that we can store in another string orvariable.
- Command substitution provides the capability to convert the result of a command to astring.
- The command substitution operator that converts the output of a command to a string **is a dollar sign and a set of parentheses.**
- To invoke the command substitution, we enclose the command in a set of parentheses preceded by dollarsign($)
- When we use this command substitution, the command is executed and output is created and then converted to string ofcharacters.
- Ex: simple demonstration of commandsubstitution.
  $ echo " The date and time are:date" Output:
  The date and time are :date
- Using command substitution with date- using dollar sign along with the command

enclosedin parenthesis

```
$echo “The date and time are:
$(date)“ Output
The date and time are:Mon Oct307:09:48 GMT 2016
```

7. Searching for a pattern:

7.1 grep(globally search regular expression andprint)

Unix has a special family of commands for handling search requirements and the principal member of the family is the **grep command**.

grep scans its input for a pattern and displays lines containing the pattern, the line numbers or filename where the pattern occurs.

Syntax:

grep options pattern filename(s)

grep searches for pattern in one or more filename or the standard input if no filename is specified. The first argument(barring the options) is the pattern and the remaining arguments are filenames.

Considerthreefiles emp.lst, emp1.lst, emp2.lst

$cat emp.lst

```
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
104|rajesh|manager|marketing|12/04/72|5800
105|adarsh|executive|sales|07/09/57|5300
```

$catemp1.lst

```
201|anil|director|sales|05/01/59|5000
202|sunil|director|marketing|12/06/51|6000
203|gupta|director|production|09/08/55|7000
```

$catemp2.lst

```
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
304|v.k.agrawal|director|marketing
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
401|sumith|general manager|marketing
402|sudhir agarwal|director|production
```

Ex 1: Now to search the pattern **sales** in **emp.lst** file

```
$ grep "sales" emp.lst
101|sharma|general manager|sales|10/09/61|6700
103|aggarwal|manager|sales|03/05/70|5000
105|adarsh|executive|sales|07/09/57|5300
```

Ex 2: To search the pattern director from 2 files i,e emp.lst and emp2.lst

```
grep "director" emp.lst emp2.lst
emp.lst:102|kumar|director|Sales|09/09/63|7700
emp2.lst:302|sudhir agarwal|director|production
emp2.lst:304|v.k.agrawal|director|marketing
emp2.lst:402|sudhir agarwal|director|production
```

7.2 grep along withoptions

Table 13.1 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*	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen.
-x	Matches pattern with entire line (doesn't match embedded patterns)
-f *file*	Takes patterns from *file*, one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings (in **fgrep**-style)

7.2.1 *Ignoring case:* when you look for a name but are not sure of the case, use the –i optionto ignore case for patternmatching.

```
$ grep -i "sales" emp.lst
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
105|adarsh|executive|sales|07/09/57|5300
```

7.2.2 *Deleting lines(-v):* The –v option selects all lines except those containing

```
 $ grep -v "director" emp2.lst
301|anil Agarwal|manager|sales
303|rajath Agarwal|manager|sales
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
401|sumith|general manager|marketing
```

thepattern

The lines containing the pattern **director** are deleted in the output.

7.2.3 Dislaying line numbers(-n).

The –n option displays the line numbers containing the pattern along with the line

```
$ grep -n "sales" emp.lst
1:101|sharma|general manager|sales|10/09/61|6700
3:103|aggarwal|manager|sales|03/05/70|5000
5:105|adarsh|executive|sales|07/09/57|5300
```

7.2.4 Counting lines containingpatter(-c)

The –c option counts the number of lines containing the

```
$ grep -c "manager" emp2.lst
```

pattern. 4

7.2.5 Dislayingfilenames(-l)

The –l option displays only the names of the files containing the pattern. Here the pattern **manager** is searched in all files ending with .lst (*.lst)

```
$ grep -l "manager" *.lst
emp2.lst
emp.lst
```

7.2.6 Matching multiple patterns(-e)

By using –e option, you can match multiple patterns

```
$ grep -e "agarwal" -e "Agarwal" -e "agrawal" emp2.lst
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
304|v.k.agrawal|director|marketing
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
402|sudhir agarwal|director|production
```

7.3 BASIC REGULAREXPRESSIONS

Table 13.2 The Basic Regular Expression (BRE) Character Subset

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*
[^a-zA-Z]	A nonalphabetic 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

7.3.1 The character class[]

A regular expression lets you specify a group of characters enclosed within a pair of rectangle brackets [ ], in which the match is performed for a single character in the group.

Thus the expression

[aA] **Matches either a orA**

```
$ grep "[aA]garwal" emp2.lst
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
402|sudhir agarwal|director|production
```

7.3.2 Negating aclass(^)

Regular expressions use the **caret(^)** to negate the character class, while the shell uses **bang(!) Ex:**

[^a-zA-Z] matches a non-alphabeticcharacter

7.3.3 The*(asterisk)

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

The patttern g*

Matchesnone, g, gg, gg, ggg,.....

```
$ cat file.lst
ourrences of a character
ocurrences of a character
occurrences of a character
occcurrences of a character

$ grep "oc*urrences" file.lst
ourrences of a character
ocurrences of a character
occurrences of a character
occcurrences of a character
```

7.3.4 The dot (.)

A .matches a single character where as the shell uses **?** to indicate that.

```
$ grep "10." emp.lst
101|sharma|general manager|sales|10/09/61|6700
102|kumar|director|Sales|09/09/63|7700
103|aggarwal|manager|sales|03/05/70|5000
104|rajesh|manager|marketing|12/04/72|5800
105|adarsh|executive|sales|07/09/57|5300
```

Here the .matches single character. It list all files beginning with 10 followed by single character. It displays lines with id 101, 102, 103, 104, 105.

7.3.5 Specifying pattern locations(^ and$)

The two regular expressions characters that match pattern at the beginning or end of line .

^(caret) Matching at the beginning of the line

$(dollar) Matching at the end of the line

```
$ grep "^3" emp2.lst
301|anil Agarwal|manager|sales
302|sudhir agarwal|director|production
303|rajath Agarwal|manager|sales
304|v.k.agrawal|director|marketing
305|v.s.agrawal|deputy manager|sales
306|agrawal|executive|sales
```

^3 matches all the lines beginning with digit 3.

```
$ grep "5...$" emp.lst
103|aggarwal|manager|sales|03/05/70|5000
104|rajesh|manager|marketing|12/04/72|5800
105|adarsh|executive|sales|07/09/57|5300
```

5...$ matches all the lines ending with four digit number beginning with 5.

7.4 EXTENDED REGULAREXPRESSIONS(ERE)

ERE make it possible to match dissimilar patterns with a single expression.The ERE has to be used with **–E option.**

Table 13.3 The Extended Regular Expression (ERE) Set Used by **grep, egrep** and **awk**

Expression	*Significance*
ch+	Matches one or more occurrences of character *ch*
ch?	Matches zero or one occurrence of character *ch*
exp1|exp1	Matches *exp1* or *exp2*
GIF|JPEG	Matches GIF or JPEG
(x1|x2)x3	Matches *x1x3* or *x2x3*
(lock|ver)wood	Matches lockwood or verwood

7.4.1 The + and?

+ Matches one or more occurrences of the previouscharacter
? Matches zero or one occurrences of the previouscharacter.

```
$ grep -E "oc+urrences" file.lst
ocurrences of a character
occurrences of a character
occcurrences of a character
```

The + symbol matches one or more occurrences of characterci,e **c, cc, ccc**
The occurrences of a character is not matched by +c , since there is no c in the occurrences.

```
$ grep -E "oc?urrences" file.lst
ourrences of a character
ocurrences of a character
```

The ?symbol matches zero or one occurrences of characterci,e **0 c , 1c.**
The other two lines occurrences (2 c) and occurrences (3 c) are not matched.

7.4.2 Matching multiple patterns(|, (and))

- The | is the delimiter for the multiplepatterns.
- Consider the file ere.lst with two lines of employeedetails.

```
$ cat ere.lst
235|barun sengupta|director|sales
279|s.n. dasgupta|manager|marketing
```

- To locate both sengupta and dasgupta from fileere.lst

```
$ grep -E "sengupta|dasgupta" ere.lst
235|barun sengupta|director|sales
279|s.n. dasgupta|manager|marketing
```

- The characters (and) lets you group patterns and use of | inside the parenthesis, you canframe more compactpatterns.

```
$ grep -E "(sen|das)gupta" ere.lst
235|barun sengupta|director|sales
279|s.n. dasgupta|manager|marketing
```

4. ORDINARY AND ENVIRONMENTVARIABLES

A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.

4.1VARIABLENAMES

- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type ofdata.
- A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.
- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
- By convention, Unix Shell variables would have their names inUPPERCASE.
- The following examples are valid variable
  names – VAR_1
  VAR_2
  TOKEN _A

4.2 DEFININGVARIABLES:

- Variables are defined as follows–
- variable_name=variable_va
  lue Forexample:
  NAME="Sumitabha Das"

4.3 ACCESSINGVARIABLES:

- To access the value stored in a variable, prefix its name with the dollar sign ($) –
- For example, following script would access the value of defined variable NAME and would print it on STDOUT–

```
#!/bin/sh
NAME="Sumitabha
Das" echo $NAME
```

4.4 ENVIRONMENT VARIABLES–

An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

SHELL: points to the shell defined as default.

DISPLAY : Contains the idenifier for the display that X11 programs should use by default.

HOME: Indicates the home directory of the current user; the default argument for the cd built in command

IFS: Indicates the Internal Field Separator that is used by the parser for word splitting after expansion.

PATH : Indicates search path for commands.It is a colon separated list of directories in which the shell looks for commands.
PWD: Indicates the current working directory as set by the cd command.
RANDOM: Generates a random integer between 0 and 32767 each time it is referenced.
SHLVL: Increments by one each time an instance of bash is created.
UID: Expands to the numeric user ID of the current user initialized at shell prompt.

- Following is the sample example showing few environment variables–

```
$ echo $HOME
/root
]$ echo $DISPLAY

$ echo
$TERM xterm
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/amrood/bin:/usr/local/bin
$
```

PS1(Prompt String one) and PS2 Environment Variables

The characters that the shell displays as your command prompt are stored in the variable PS1. You can change this variable to be anything you want. As soon as you change it, it'll be used by the shell from that point on.
For example, if you issued the command –

```
$PS1='=>'
=>
=>
```

Your prompt would become =>.
When you issue a command that is incomplete, the shell will display a secondary prompt and wait for you to complete the command and hit Enter again.
The default secondary prompt is > (the greater than sign), but can be changed by re-defining the **PS2** shell variable –
Following is the example which uses the default secondary prompt –

```
$ echo "this is a
> test"
this is
a test
$
```

```
$PS= '-->'
$ echo "this is a
-->test"
```

4.5 The .profileFile

- The file **/etc/profile** is maintained by the system administrator of your UNIX machine and contains shell initialization information required by all users on asystem.
- The file **.profile** is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configureincludes
- The type of terminal you are using
  A list of directories in which to locate commands
  A list of variables effecting look and feel of your terminal.
- You can check your **.profile** available in your home directory. Open it using **vi**editor and check all the variables set for yourenvironment.

4.6 SHELLSCRIPTS

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

Structure of shell script:

```
#!/bin/sh
# script.sh: Sample shell
script echo " Todays date:
`date`" echo "This
month calendar" cal
`date` "+%m 20%y" echo
"My Shell:$SHELL"
```

output:

```
$sh script.sh
Todays date : Mon Nov 7 10:03:42 IST
2016 This month's calendar:
November 2016
```

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

```
My shell: /bin/sh
```

- Use your vi editor to create the shell scriptscript.sh.
- The script runs three echo commands and shows the use of variable evaluation and command substitution. It also prints the calendar of the currentmonth.
- Note that the # is comment character, that can be placed anywhere in a line; the shell ignores all characters placed on itsright.
- However this doesnot apply to the first line, which also begins with a #. This is interpreter line that was mentionedpreviously.
- It always begins with #! and is followed by pathname of the shell to be used for

running the script. However this line specifies the bourneshell.

- To run the script, make it executable first and then invoke the scriptname

$chmod a+x script.sh
$sh script.sh

- Shell scripts are executed in a separate child shell process and this sub shell need not be of the same type as your login shell.By default child and parent shell belongs to the same type, but you can provide a interpreter line in the first line of the script to specify a different shell for yourscript.

4.7 read and readonly commands.

read:MAKING SRIPTSINTERACTIVE

- The read statement is the shell internal tool for taking the input from the user ie making scripts interactive.
- It is used with one or more variables. Input is supplied through the standard input is read into these variables.
- When you use statement
  like: read name
  the script pauses at that point to take input from the keyboard. whatever you enter is stored in the variable name. since this is a form of assignment , no $ is used before the name.
- A single read statement can be used with one or more variables to let you enter multiple arguments.
  read pname flname
- The script asks for a pattern to be entered. Input the string director, which is assigned to the variable pname. Next the script asks for the filename enter the string emp.lst which is assigned to the variable flname.
- grep runs with these two variables asarguments

```
#!/bin/
sh
#emp1.s
h #
echo "Enter the pattern to be searched
 :\c" readpname
echo "  Enter the file to be used
 :\c" readflname
echo "  Searching for $pname from file $flname"
grep "$pname"$flname
echo "Selected rows shown above"
```

Output:
$sh emp1.sh
Enter the pattern to be
searched:**director**Enter the file to be
used: **emp.lst** Searching for **director**
from file emp.lst
101
sharma|**director**|production|12/03/70|7000
102|barun|**director**|marketing|11/06/67|7800
selected rows shown above

4.8 USING COMMAND LINEARGUMENTS

- When arguments are specified with a shell script they are assigned to certain special variables- positional parameters.
- $*→ store the complete set of positional parameters as a singlestring.
- $#-->It is set to the number of argumentsspecified.

- $0- → holds the command nameitself.
- When arguments are specified in this way the first word (the command itself) is assigned to $0, the second word(the first argument) to $1, the third word(the second argument) to$2.

```
#!/bin/
sh
#emp2.
sh #
echo "Program:$0
The number of arguments specified is $#
The arguments are $*"
grep "$1" $2
echo "\n job over"
```

Output:

```
$ sh emp2.sh directoremp.lst
Program: emp2.sh
The number of arguments
specifiedis:2The
argumentsaredirector        emp.lst
101|
sharma|director|production|12/03/70|700
0
102|barun|director|marketing|11/06/67|78
00 job over
```

4.9 SPECIAL PARAMETERS USED BY THESHELL.

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

4.10 Exit and exit status ofCommand.

- The shells exitcommand
  exit 0 Used when everything went fine.
  exit 1 Used when something went
  wrong

Its through the exit command or function that every command returns an exit status to the caller.

Further a command is said to return true exit status if it executes successfully and false if its fails.

- THE PARAMETER $? :It 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 If no exit status is specified then $? is set tozero(true).

- Consider two files file1 which exist in current directory and file2 which does notexist

$ ls –l file1;echo$? **/*file1 attributes arelisted**
Output :0 /*exit status $?=0, since cmd executedsuccessfully

$ ls –l file2;echo$? **/*error since file2 doesnotexist**
Output: 1 /*exit status $?=1, since cmd executionfailed.

4.11 THE LOGICAL OPERATORS && and || - CONDITIONALEXECUTION

- The shell provides two operators that allow conditional execution.the && and||.
- The syntax:

cmd1 &&
cmd2 cmd1 ||
cmd2

- Consider a file emp.lst

$cat emp.lst
1066| sharma | **director** |sales |03/09/66 |
7000 1098| Kumar |**director**|
production|0/08/67 | 8200 1082|sumith|
manager|marketing|09/09/73| 7090

- The && delimits two commands ; the command cmd2 is executed only when cmd1succeeds.

$ grep "director" emp.lst && echo "Pattern found in file"
Output:
1066| sharma | **director** |sales |03/09/66 |
7000 1098| Kumar |**director**|
production|0/08/67 | 8200 Pattern found in
file

- The || operator plays inverse role. The second command is executed only when the firstfails.

$grep " deputy manager" emp.lst || echo "Pattern not found"
Output:
Patternnotfound /* cmd1 - **deputy manager** is not found inemp.lst. Hence cmd1 fails. Therefore cmd2 "**pattern not found**" executes.

$grep "manager" emp.lst || echo "Pattern not found"
Output
1082|sumith|**manager**|marketing|09/09/73|709 /*Here cmd1 is executed successfullyi,e manager is found ,therefore cmd2 will not be executed.

4.12 CONDITIONALSTATEMENTS:

4.12.1 If 4.12.2case

4.12.1 The ifCONDITIONAL

If command is successful then	If command is successful then	If command is successful then

execute commands else execute commands fi	execute commands fi	execute commands elif command is successful then .. else .. fi

The **if**statement makes two way decision making depending on the fulfillment of a certain condition.

- **If** also requires a**then.**
- It evaluates the success or failure of the command that is specified in its command line. If command succeeds the sequence of the commands following it is executed. If commands fails then the **else** statement isexecuted
- Every **if**is closed with corresponding with**fi.**

```
#!/bin/
sh
a=10
b=20
if [ $a==$b
 ] then
echo "a is equal to
b" elif [ $a –gt
$b  ] then
echo "  a is greater than
b" elif [ $a -lt $b]
then
echo "  a is lesser than b"
else
echo "  None of the conditions
met" fi
```

output:
a is lesser than b

4.12.2 The caseCONDITIONAL

- The case statement is the second conditional offered by theshell
- The statement matches an expression for more than one alternative and uses a compact construct to permit multiwaybranching.

The general syntax is

```
case expression in
pattern1 )
commands1 ;;
pattern2 )
commands2 ;;
pattern3 )
commands3 ;;
……..
esac
```

case first matches expression with pattern1. If the match succeeds, then it executes commands1, which may be one or more commands. If the match fails, then pattern2 is matched and so on….Each command list is terminated with a pair of semicolons and the entire construct is closed with esac.

```
#!/bin/
sh
#menu.
sh
# echo " MENU \n
1.List of files\n 2.Processes of user\n 3.Todays
date\n 4.Users of system\n 5.Quit\n
Enter your option: \"
```

```
read choice
case "$choice"
in 1 ) ls –l ;;
2 ) ps –f;;
3 ) date;;
4 ) who ;;
5 ) exit ;;
* ) echo "invalid option"
esac
```

To run the program:

$ sh menu.sh

Output:

```
    MENU
1. List of files
2. Processes ofuser
3. Todays date
4.Users
ofsystem
5. Quit
Enter your option : 3
Sun Nov 6 18:03:06 IST 2016
```

4.12.2.1 Matching multiplepatterns:

- case can also specify same action for more than onepattern.
- For example the expression y|Y can be used to match y in both upper and lower caseletters.

```
echo "Do you wish to continue?
 :\c" readanswer
case "$answer"
in y|Y)    ;;
n|N ) exit ;;
esac
```

4.12.2.2 Wild cards: case usesthem

- case has a string matching feature that uses wildcards.
- It uses the filename matching meta characters *, ?and the character class but only to match strings but not the files in the currentdirectory.

```
case "$answer"
in [yY][eE]* )  ;;
[nN][oO]  ) exit;;
* ) echo "Invalid response"
esac
```

4.13 USING test and [] to evaluate theexpressions.

Test uses the certain operators to evaluate the condition on its right and returns either true or false exit status which is then used by if for making decision .

Test works in 3 ways:

4.13.1. Compares two numbers (NUMERICCOMPARISION)
4.13.2. compares two strings or a single one for a null value.(STRINGCOMPARISION)
4.13.3. checks a file attributes. (FILETEST)

4.13.1. NUMERICCOMPARISION:

The numeric comparison operators used by test are

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

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 $?
```

Output : 1

```
$ test $x –lt $y ; echo $?
```

Output: 0

```
$ test $z –gt $y ; echo $?
```

Output: 1

4.13.2. STRINGCOMPARISION

testcan be used to compare strings with yet another set of operators.

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 null string
Stg	String stg is assigned and not null
s1==s2	String s1= s2(Korn and bash only)

Example:

```
#!/bin/
sh
a="abc"
b="efg"
if [ $a = $b ]
then
echo "a is equal to
b" else
echo "a is not equal to
b" fi
```

output:
a is not equal to b

4.13.3. FILETESTS

testcan be used to test the various file attributes like its type(file, directory or symbolic link) or its permissions(read,write,execute)

Test	True if File
-f file	File exists and is regular file
-r file	File exist and is readable
-w file	File exists and is writable
-x file	File exists and is execuatble
-d file	File exists and is a directory
-s file	File exists and has a size greater than zero
-e file	File exists (Korn and bash only)
-L file	File exists and is symbolic link
f1 -nt f2	f1 is newer than f2(Korn and bash only)
f1 -ot f2	f1 is older than f2(Korn and bash only)
f1 -ef f2	f1 is linked to f2(Korn and bash only)

```
$ls –l emp.lst
-rw-rw-rw-  1      kumar      group       870    Sep8 15:52    emp.lst

$ [ -f emp.lst ] ; echo $?              //  -f ( emp.lst file exist and its regular file)
0                                        //Yes
$ [ -x emp.lst ] ; echo$?              //  -x  (emp.lst file is executable ornot)
1                                        //No
```

4.14 while Looping

- The while statement repeatedly performs a set of instructions until the control command return a true exit status.
- The general syntaxis

  while*condition* **is**
  true do
  command
  s done
- The commands enclosed by do and done are executed repeatedly as long as condition remains true.

4.14.1 Using while to wait for afile

- There are situations when a program needs to read a file that is created by another program, but it has to wait until the file is created.
- The script, monitfile.sh periodically monitors the disk for the existence of the file. And then executes the program once the file has been located.
- It makes use of the external sleep command that makes the script pauses for the duration in seconds as specified in its arguments.
- The loop executes repeatedly as long as the file invoice.lst cannot beread.
- If the file becomes readable the loop is terminated and the program alloc.pl isexecuted.
- We use the sleep command to check every 60 seconds for the existence of thefile.

4.14.2 Setting up an infiniteloop

Suppose you as system administrator want to see the free space available on your disk every five minutes

```
while
true do
df –t
sleep
300
done &
```

df reports free space on disk . sleep command is used to hek for every 300 seconds(5 minutes) . & after done runs loop in background

---

4.15 for : LOOPING WITH ALIST

The shells for loop differs in structure from the ones used in other programming languages. There is no three part structure.

```
for variables in list
do
command
s done
```

The loop body also uses the keyword do and done. But the additional parameters here are variable and list. Each whitespace separated word in list is assigned to variable and commands are executed until list is executed .

Ex:

```
$for file in chap20 chap21
 chap22 do
 cp $file{$file}.bak
 echo $file copied to
 $file.bak done
```

Output:
chap20 copied to chap20.bak chap21 copied to chap21.bak chap22 copied to chap22.bak

POSSIBLE SOURCES OF THE LIST

4.15.1 List from variables:

You can use series of variables in the command line. They are evaluated by the shell before executing the loop

```
$ for var in $PATH
$HOME do
echo “$var”
done
```

Output:

```
/bin:/usr/bin:/home/local/bin     /*$var=$PATH
/home/henry                       /*$var=$HOME
```

4.15.2 List from command substitution

The following for command line picks up its list from clist.

```
for file in `cat clist`
```

4.15.3 List from wild cards

When the list consists of wild cards, the shell interprets them as filename.

```
for file in *.htm
*.html do
gzip
$file
done
```

4.15.4 List from positionalparameters

foris also used to process positional parameters that are assigned from command line arguments it uses the shell parameter $@ to represent all command line arguments

```
for pattern in "$@"
```

4.15.5 basename: changing filenameextensions

- When basename is used with two arguments it strips off the second argument from the first argument

```
$ basename note.txttxt
note.                    //txt strippedoff
```

- Used to rename filename extensions from txt to doc

```
for file in
*.txt do
leftname= `basename $file txt
` mv $file${{leftname}doc
done
```

- If for picks up note .txt as the firstfile,
- Leftname stores note.(withdot)
- mv simply adds doc to the extractedstring(note.)

4.16 set and shift: MANIPULATING THE POSITIONALPARAMETERS

- set assigns its argument to positional parameters $1,$2 and soon.

```
$set 989 878779
$_
```

This assigns the value 989 to the positional parameter $1, 878 to the positional parameter $2 and 779 to $3

Ex:

```
$echo "\$1 is $1, \$2 is $2, \$3 is $3"
```

Output: $1 is 989, $2 is 878, $3 is 779

```
$echo "The $# arguments are $*"
```

Output: The 3 arguments are 989 878779

Shift : Shifting Arguments left

Shift transfers the contents of a positional parameter to its immediate lower numbered one.

```
$ set `date'
$echo "$*
```

Output:Wed Nov 9 09:04:30 IST 2016

$shift
$ echo $1$2 $3 $4 $5
Output: Nov 9 09:04:30 IST 2016

$shift 2
$echo $1$2 $3
Output:09:04:30 IST 2016

4.17 The HERE DOCUMENT(<<)

- The shell uses << symbols to read data from the same file containing thescript.
- This is referred to as here document , signifying that the data is here rather than in a separate file.
- If the message is short you can have both the command and message in the samescript. mail sharma <<MARK
  Your program for printing the invoices has been
  executed on `date`. The updated file is $flname
  MARK
- The here document symbol(<<) followed by three lines of data and a delimiter (the string MARK)
- The shell treats every line following the command and delimited by MARK as input to the command.
- Sharma at the other end will see the three lines of message text with the date inserted by command substitution and the evaluatedfilename.

4.18 trap: INTERRUPTING APROGRAM

- By default shell scripts terminate whenever the interrupt key is pressed. It may leave a lot of temporary files on disk.
- The trap statement lets you do things you want in case the script receives a signal. The statement is normally placed at the beginning of a shell script and uses twolists
  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 of one or moresignals.

  trap**'rm $$* ; eho "Program Interrupted" ; exit '** HUP INT TERM
  trap 'cmd_list' sig_list
- trap is a signalhandler.
- Here it first removes all the files expanded from $$*, echoes a message and finally terminates the script when the signals SIGHUP(1), SIGINT(2), SIGTERM(15) are sent to the shell process running thescript.
- When the interrupt key is pressed it sends the signal number2.
