UNIX

Concepts and Applications

A. Yugandhar Reddy

PUBLISHED BY PUBLISHER

BOOK-WEBSITE.COM

First printing, March 2016

Contents

II Part Two

Part One

1. General Purpose Commands

1.1 General Purpose Utilities

1.	`cal`	print current month calender
2.	`cal 1947`	print 1947 calender
3.	`cal 8 1947`	print 1947 august calender
4.	`date`	display system date
5.	`date +%D`	dd/mm/yy
6.	`date +%d-%m-%Y`	dd-mm-yyyy
7.	`echo "Enter:"`	by default cursor goes to next line
8.	`echo "Enter: \c"`	cursor stays in current line (carryt return)
9.	`xcal`	display graphical calculator
10.	`bc`	command line calculator
	`12+5`	
	17	
	`12*12`	
	144	
	`2^10`	
	1024	
	`9/5`	decimal portion truncated
	1	
	`scale=2`	truncate to 2 decimal places
	2.42	
	`ibase=2`	
	1010	
	10	
	`obase=2`	
	14	
	1110	

11.	`script`	recored session to the file *typescipt* by defalut
12.	`script log`	recored session to the file *log*
13.	`script -a`	append to existing file *typescript*
14.	`passwd`	change the password of current user.
15.	`setterm -term linux -back blue -fore yellow -clear` black|blue|green|cyan|red|magenta|yellow|white|default	changing the terminal color
16.	`who`	prints information about current user information like *user,terminal associated to user,login time*
17.	`who -Hu`	displays logged in users with headers
18.	`who -b`	boot time of the system
19.	`who -d`	prints dead process
20.	`who am i`	displays the current user details
21.	`uname`	name of the operating system
22.	`uname -a`	print all information of system os name,host name,processor,hardware name,kernal version,kernal release
23.	`uname -r`	version of os
24.	`uname -n`	hostname
25.	`tty`	file name of the terminal your using
26.	`alias ls="ls -l"`	
27.	`alias cp="cp -i"`	
28.	`alias rm="rm -i"`	
29.	`alias`	display all aliases
30.	`unalias ls cp`	remove alias
31.	`env`	display all environment variables
32.	`sh`	create new shell
33.	`history`	
34.	`history -5`	last 5 commands
35.	`cd $_`	using arguments of previous commands
36.	`set`	display variable in current shell

1.2 The File System

1.2.1 cd,pwd

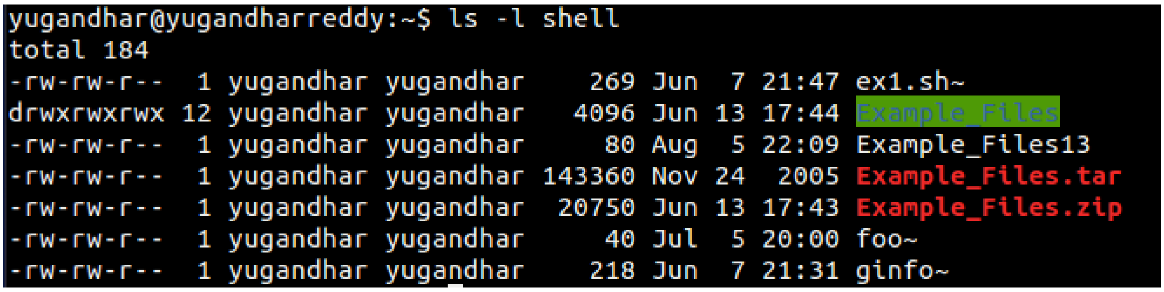
1.	`echo $HOME`	home directory of current user
2.	`pwd`	present working directory
3.	`cd ..`	change to parent directory
4.	`cd .`	change to the current directory
5.	`cd /`	change to root
6.	`cd ~`	change to home directory
7.	`cd`	cd with out arguments change to home directory

1.2.2 ls:Listing Files

1.	`ls`	list file in the pwd
2.	`ls -x`	output in multiple columns
3.	`ls -a`	list all (hidden files also)
4.	`ls -F`	identify directories and executable files
5.	`ls -x helpdir progs`	list the directories helpdir and progs
6.	`ls -R`	list files and sub-directories recursively
7.	`ls -i`	display inode number
8.	`ls -d dir1 dir2`	

Note: when ever you use `-d` option arguments should be directory(s)

9. `ls -t` sort by last modification time
10. `ls -u` sort by last access time
11. `ls -l shell` prints seven attributes of the files



```
yugandhar@yugandharreddy:~$ ls -l shell
total 184
-rw-rw-r--  1 yugandhar yugandhar    269 Jun  7 21:47 ex1.sh~
drwxrwxrwx 12 yugandhar yugandhar   4096 Jun 13 17:44 Example_Files
-rw-rw-r--  1 yugandhar yugandhar     80 Aug  5 22:09 Example_Files13
-rw-rw-r--  1 yugandhar yugandhar 143360 Nov 24  2005 Example_Files.tar
-rw-rw-r--  1 yugandhar yugandhar  20750 Jun 13 17:43 Example_Files.zip
-rw-rw-r--  1 yugandhar yugandhar     40 Jul  5 20:00 foo~
-rw-rw-r--  1 yugandhar yugandhar    218 Jun  7 21:31 ginfo~
```

Figure 1.1: ls -l shell output

output of the above command is explained bellow

-	rw-	rw-	r- -	1	yugandhar	yugandhar	269	Jun 7 21:47	ex1.sh
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)

(a) type of the file, - for regular files, `d` directories
(b) permissions of owner
(c) permissions of group
(d) permissions of others
(e) Number of links in case of files,Number of entries in the directory in case of directory
(f) owner
(g) group
(h) file size in bytes
(i) last modified time
(j) file or directory name

1.3 Handling Ordinary files

1.3.1 `cat,cp,rm,mv,cmp,comm,diff,wc,file`

1. `cat emp.lst` display file in the terminal
2. `cat -v emp.lst` display non printable characters
3. `cat -n emp.lst` display line numbers
4. `cat > foo` terminal wait for text to be write into file. EOF is [*ctrl+d*]
5. `cp foo goo`
6. `cp file1 file2 file3 dir` while copying multiple files last arguments should be a directory
7. `cp -i foo goo` interactive copy
8. `cp -R progs newprogs` copy directory structures
9. `rm chap1 chap2`
10. `rm *` all files gone (think before use)
11. `rm -i foo` intractive remove
12. `rm -R *` remove every thing by recursive traversal

Table 1.1: Abbrviations used by `chmod`

Category	Operations	Permission
u-User	+ Assign permission	r-4- Read
g-Group	- Remove permission	w-2- Write
o-Other		x-1- Execute
a-All(ugo)		

13. `rm -f goo` force remove if file is write protected
14. `mv foo goo` rename
15. `mv dir1 dir2` directory rename
16. `mv foo goo hoo dir1` moving files into directory
17. `cmp chap01 chap02` two files are compared byte by byte first mismatch is returned to terminal. silence if no mismatch
18. `comm chap01 chap02` three column output 1. uniq to 1st file, 2. uniq to second file, 3. common to both
19. `comm -13 chap01 chap03` drop fist and third columns in the output
20. `diff chap01 chap03` instructions indicate the changes that are required to make two files identical
21. `wc emp.lst` counts lines,words, characters
22. `wc -l emp.lst` count lines
23. `wc -w emp.lst` count words
24. `wc -c emp.lst` count characters
25. `wc emp.lst tem.lst` count both files
26. `file emp.lst` display file type
27. `file *` display file type of all file in pwd

1.4 File Ownership and File Permission

1. *id* shows user id and groud id
2. `chmod u+x emp.lst` assigning execute permissions to user (owner)
3. `chmod +x emp.lst` assigning execute permissions to all (default)
4. `chmod ugo+x emp.lst` assigning execute permissions to user,group and others.
5. `chmod a+x emp.lst` assigning execute permissions to all
6. `chmod 742 emp.lst` assign rwx → user, r - - → group, - w - → other
7. `chmod 210 emp.lst` assign - w - → user, - - x → group, - - - → other
8. `chmod -R 742 htdocs` assign permissions recursively to directory tree
9. `chown` *`sekhar`* `emp.lst` changeing the owner of emp.lst to *sekhar*
10. `chgrp` *`dba`* `emp.lst` changeing the ground of emp.lst to *dba*
11. `chown sekhar:dba emp.lst` changeing owenr and group in a single line.

1.5 The Shell

1. `cat >foo 2>goo` output redirection to foo,error redirection to goo
2. `rm -f *` forcefull removel of all files
3. `date | cut -d " " -f 1` pipeline output
4. `comm file[12]` comm file1 file2
5. `ls chap*`
6. `echo *` display all files in the pwd
7. `rm *.o` very dangerous

8. `ls chap?`
9. `ls chap??`
10. `ls .???*` .profile like files listed
11. `ls emp*lst`
12. `ls chap0[123]`
13. `ls *.[!co]` all files with a single character extension but not .c and .o
    Note: to remove special meaning of shell symbols use Escaping and single quoting

14. `ls chap0\[1-3\]`
15. `echo '\'`
16. `rm 'chap*'`
17. `rm My Document.doc`
18. `rm "My Document.doc"`
19. `cat file » foo 2» goo` appending
20. `who | tee user.txt` `tee` writes to both file and terminal
21. ``echo "today’s date is `date`"`` command substitution
22. ``echo 'today’s date is `date `'`` command substitution meaning is removed
23. .

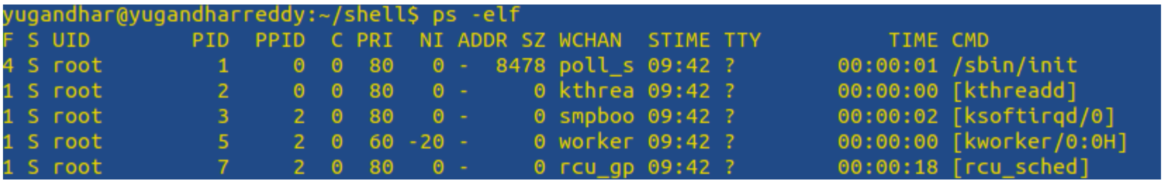
```
$ total=5                                  no space on either side of =
$ echo $total
5
$ file=foo;ext=.c
$ full=$file$ext
foo.c
$ count=1
$ readonly count
$ unset count                              can not uset readonly variable
```

1.6 The Process

1.6.1 ps,nice

1. `echo $$` print PID of current loging shell
2. `ps` displays the process owned by the current user
3. `ps -e` all process including system and user processes
4. `ps -f` full list UID,PID,PPID,C,STIME,TTY,TIME,CMD
5. `ps -u yugandhar` list process associated with the user yugandhar
6. `ps -a` processes of all users excluding processes not associated with the terminals
7. `ps -l` long list showing memory related inforamation
8. `ps -t term` process runing on the terminal
9. `ps -elf`

- **F** Flags associated with the process.
  1- forked but didnot executed
  4- used super-user privileges
- **S** Minimum state display.Process state
- **C** CPU utilization
- **PRI** Priority; Heigher number lower priority
- **NI** 19-nice; -20 not nice



```
yugandhar@yugandharreddy:~/shell$ ps -elf
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  0  80   0 -  8478 poll_s 09:42 ?        00:00:01 /sbin/init
1 S root         2     0  0  80   0 -     0 kthrea 09:42 ?        00:00:00 [kthreadd]
1 S root         3     2  0  80   0 -     0 smpboo 09:42 ?        00:00:02 [ksoftirqd/0]
1 S root         5     2  0  60 -20 -     0 worker 09:42 ?        00:00:00 [kworker/0:0H]
1 S root         7     2  0  80   0 -     0 rcu_gp 09:42 ?        00:00:18 [rcu_sched]
```

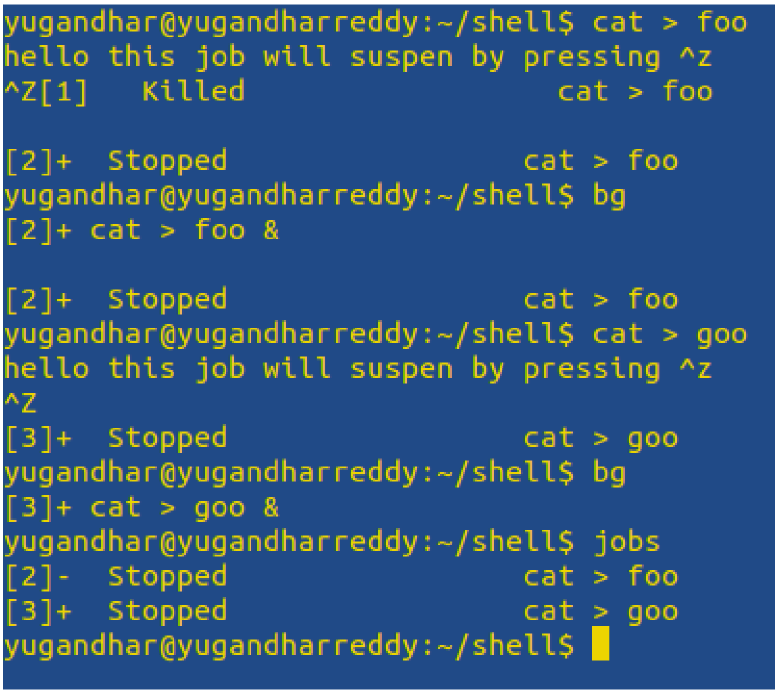
Figure 1.2: `ps -elf`

- **WCHAN** Address of the kernal function where process is sleeping
- `STIME` Start time of process
- **SZ** Size of the process
- **PID** Process ID
- **PPID** Parent process ID
- **TTY** ?- means system process
- **TIME** accumulated cpu time, user + system. The display format is usually "MMM:SS", but can be shifted to the right if the process used more than 999 minutes of cpu time.

10. `nice wc -l emp.lst` run job with lower priority
11. `nice -n 5 wc -l emp.lst` nice value is increased to 5

1.6.2 Job Control(&,nohup,kill,bg,fg)

Suspend the job by pressing *[Ctrl-z]*. And use `bg` command to send suspended job to background.Example is showen bellow.



```
yugandhar@yugandharreddy:~/shell$ cat > foo
hello this job will suspen by pressing ^z
^Z[1]   Killed                  cat > foo

[2]+  Stopped                 cat > foo
yugandhar@yugandharreddy:~/shell$ bg
[2]+ cat > foo &

[2]+  Stopped                 cat > foo
yugandhar@yugandharreddy:~/shell$ cat > goo
hello this job will suspen by pressing ^z
^Z
[3]+  Stopped                 cat > goo
yugandhar@yugandharreddy:~/shell$ bg
[3]+ cat > goo &
yugandhar@yugandharreddy:~/shell$ jobs
[2]-  Stopped                 cat > foo
[3]+  Stopped                 cat > goo
yugandhar@yugandharreddy:~/shell$ 
```

Figure 1.3: `job control`

1. `gedit sort.c &` return PID and run in background. No logout
2. `nohup gedit sort.c &` return PID and run in background. logout allowed
3. `jobs` display the jobs running background

4. `fg` bring most recent job to foreground
5. `fg %1` bring first job to foreground
6. `fg %sort` bring *sort* job to foreground
7. `fg %?cat` bring a job containing string *cat* to foreground
8. `bg %2`
9. `bg %?cat`
10. `kill 107` kill the process with PID 107
11. `kill 107 4004 4123`
12. `kill $!` kill last background job
    Some time kill signal is ignored by the some process. in that case SIGKILL(9) is used.SIGKILL(15) is default

13. `kill -s KILL 121` Recommended way of kill
14. `kill -9 121` same as aboce but not recommended

1.7 Job Schduling (at,batch,crontab)

1. `at`:One-Time Execution

   ```
   $ at 14:25
   at>grep.sh
   at>sed.sh
   [Ctrl-d]
   command will be executed using /use/bin/bash
   job 10411888880.a at Wed Aug 24 14:25:00 2016
   ```

   output and errors of the schduled jobs are mailed to user mail.To avoid this you can use redirection operation

   `at> grep.sh >goo 2>foo`

   at command to uses following time formats

   at 15
   at 5pm
   at 3:08pm
   at noon *At 12:00 hours today at now + 1 year*
   at 3:08pm + 1 day
   at 15:08 August 28, 20016
   at 9am tomorrow
2. `batch < empawk1.sh` commands will executed at some convenient time.
   Time Decided by System Algorithm

3. `at -r`
   If command is sechduled with `batch` then the job is moved to special `at` queue. From where it can be removed with `at -r`
4. **Running Job Periodically**

   ```
   $ cat > cron.txt
   00-10   17   *   3,6,9   5   echo $SHELL > cronout.txt
   ```

   - `00-10` minutes of the hour;00-10 each minute in the 17th hour.
   - `17` Time of the day
   - `*` Day of the month to run.`*` Every day
   - `3,6,9` Months to run

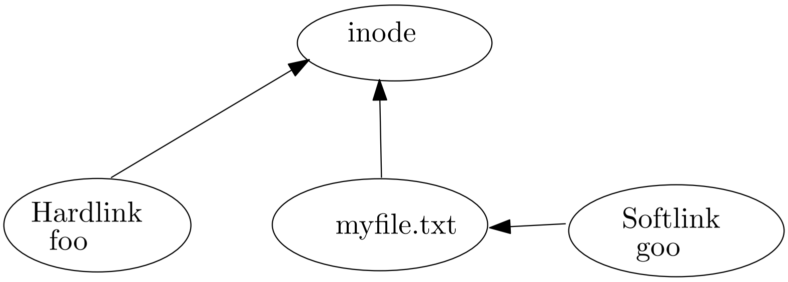
   - *5* Day of the week to run 0-sunday;5-Friday
   - *echo $SHELL > cronout.txt* Command to run
5. `crontab -l` List cron commands
6. `crontab -ir` remove cron commands in interactive mode
7. `crontab -e` edit cron commands
8. `time empawk.sh` prints time taken by the commands

1.8 Links and Others

HDD has splits each split has separate file system.Main file system is *root file system*.When system boots up other file system are mount (attach) to the root file system.

1.8.1 Hardlinks and Softlinks

1. `ln jab.mkv ./link/jab`
   Now jab.mkv and ./link/jab are pointing to the same *inode*. So their last accessed time,inode,size are identical.The file in link directory should not exist before.
2. `ln foo goo`
3. `rm goo` logical link is removed but node inode.
4. `ln foo goo`
5. `ln -f hoo goo` old link is removed and new link is created (force remove)
6. `ln foo goo hoo dir` linking multiple files to a directory. Last argument should be a directory
7. **Limitations of Hardlinks**
   - You can not link two files in two different file systems.
   - You can not link two directories even with in the same file system.
8. `ln -s foo goo` soft link (pointer) is created goo → foo
   like shortcut in windows. If foo is deleted link become `rm` is used for removal of soft links.In the ouput put of the `ls -l goo` file szie shows 4byte,it is link size



```
ln myfile.txt foo

ln -s myfile.txt goo
```

Figure 1.4: `links`

1.8.2 umask,touch

1. `umask` displays the default permissions
2. `umask 754` sets defalut permissons
3. `umask 000` All read-write permissions on
4. `umask 666` All read-write permissions on
5. `touch`
   `touch` is used to change the time stamp of the file.i.e, last access time/modified time.

6. `touch foo` set last access and modified time to current time
7. `touch 08242109 foo` set last access and modified time to 08242109 [MMDDhhmm]
8. `touch -a 08242109 foo` set last access time to 08242109 [MMDDhhmm]
9. `touch -m 08242109 foo` set modified time to 08242109 [MMDDhhmm]
   format allowed: MMDDhhmm, YYMMDDhhmm , YYYYMMDDhhmm

1.8.3 `find`:Locationg Files

syntax: `find` *path selection_ criteria action*
path: directory structure to search
selection_ criteria : selection criteria to select a file
action: action taken on selected files

1. `find / -name a.out -print` locate all a.out files
2. `find .  -name "*.c" -print` print all .c files in pwd
3. `find .  -name "[A-Z]*" -print`
4. `find / -inum 123602 -print` print a file name pointing to inode 123602
5. `find .  -type d -print` print all directories in the pwd with their paths
   d - Directory o - Ordinary Files f - Regular files
6. `find` localtes every thing in the pwd
7. `find $HOME -perm 777 -type d -print` AND condition (an implied `-a` betwwen `-perm` and `-type`)
8. `find .  -mtime +2 -print` modified in 2 days back
   +2 more than two days -2 last two days =2 exactly two days
9. `find .  -atime +365 -print` print all the file that are not accessed for one year
10. !(Negation), `-o` OR,`-a` AND *AND is implicit*
11. `find .  !  -name "*.c" -print` print all files except .c files
12. `find /home \( -name "*.sh" -o -name "*.pl" \) -print`
13. `find .  -type f -mtime +2 -mtime -5 -ls` `-a` implied
    displays regular files that are modified in more than two days and less than 5 days
14. `find /home -size +2028 -size -8192 -print` print avoce 1MB bellow 4MB files.
15. `find .  -type f -atime +365 -exec rm {} \;`
    `-exec` action is used to run linux commands.
16. `find $HOME -type f -atime +36k -ok mv {} $HOME/safe \;`
    `-ok` action is same as `-exec` but at in interractive mode

2. Simple Filters

2.1 `head`:Displaying Begining of a File

1. `head emp.lst`	by default displays first 10 lines
2. `head -n 3 emp.lst`	displays first 3 lines
3. ``gedit `ls -t	head -n 1` ``	opens last modified file.

Note No - symbol in front of integer in `head`, but it is presented in the `tail`

2.2 `tail`: Displaying The End of a File

1. `tail emp.lst`	by default displays last 10 lines
2. `tail -n -3 emp.lst`	displays last 3 lins
3. `tail +11 emp.lst`	displays 11th line onwards (ubuntu not support this)
4. `tail -f install.log`	prints the last written messages when file growing
5. `tail -c -512 emp.lst`	copies last 512 bytes from emp.lst
6. `tail -c +512 emp.lst`	copies everything except first 512 bytes

2.2.1 `head` and `tail`

Printing line number 6 to line number 12

1. `head -n 12 emp.lst | tail -n -6`

If your using ubuntu and you want to print all the lines except first 3 lines. In this case use the following sequence of commands

```
x=`wc -l emp.lst | cut -d " " -f 1`
y=`expr $x - 3 `
tail -n -$y emp.lst
```

x contail number of lines in the file

2.3 `cut`: Slitting a File Vertically

1. `cut -c 6-22,24-32 emp.lst`	cut by columns
2. `cut -c -3,6-22,24-32 emp.lst`	must be in ascending oredr

3. `cut -d "|" -f 2,3 emp.lst` `d` sets delimiter,`f` field number.
4. `cut -d "|" -f 1,4- emp.lst`
5. `who | cut -d " " -f 1`

`Note` you need to specify one the options `-f` and `-c`. These options are not optional.

2.4 `paste` Pasting Files

`paste` is used to paste the file side by side. Default delimiter is space. To specify delimiter use `-d` option.

1. `paste cutlist1 cutlist2`
2. `paste -d "|" cutlist1 cutlist2`
3. `cut -d "|" -f -4 emp.lst | paste -d "|" - cutlist2`
4. `cut -d "|" -f 5- emp.lst | paste -d "|" cutlist1 -`

2.5 `sort`: Sorting files

1. `sort emp.lst` sort based on entire line
2. `sort -t "|" -k 2 emp.lst` `-t` delimiter, `-k` primary key, 2nd field is primary key
3. `sort -t "|" -r -k 2 emp.lst` reverse sort
4. `sort -t "|" -k 2r emp.lst` reverse sort
5. `sort -t "|" -k 3,3 -k 2,2 emp.lst` sorting on secondary key also.
   3,3 primary starts at 3rd field and ends at 3rd field.
6. `sort -t "|" -k 5.7,5.8 emp.lst`
   sort starts at 5th field 7th char and end at 5th field 8th char
7. `sort -n numfile` `-n` numerical sort
   Assume numfile has integers in each line.
8. `cut -d "|" -f 3 emp.lst| sort -u | tee desigx.lst` remove repeated lines
   `tee` writes the ouput to terminal and file
9. `sort -c emp.lst` check for sort.If sorted , output is silence
10. `sort -t "|" -c -k 2 emp.lst`
11. `sort emp.lst temp.lst` concatenate and sort
12. `sort -m foo1 foo2 foo3` foo1,foo2 and foo3 are already sorted.
    merge the in the sorted oreder

2.6 `uniq`: Locate Repeated and Non-repeated Lines

1. `cut -d "|" -f 3 emp.lst| sort | uniq -u` select only unique lines
2. `cut -d "|" -f 3 emp.lst| sort | uniq -d` select duplicate lines only once
3. `cut -d "|" -f 3 emp.lst| sort | uniq -c` frequency count

2.7 `tr`: Translating Characters

`tr` *expression1 expression2 standard input*

Note it does not file as argument

1. `tr '|/' '~-'< emp.lst` | and / are replaced by ~and - respectively
2. `tr -d '|/' < emp.lst` delete | and / chars
3. `tr -cd '|/' < emp.lst` except de | and / delete all chars
4. `tr -s ' ' < emp.lst` compressing multiple consecutive spaces
5. `cut =d "|" -f 3 emp.lst | tr '[a-z]' '[A-Z]'`

3. Filters Using Regular Expressions - `grep` and `sed`

3.1 `grep`:Search for a Pattern

Syntax: `grep` *options pattern file(s)*

- if the pattern is found. `grep` output the total line where match occurs.
- if the pattern is not found. `grep` results in silence.
- Use double quotes to quote the pattern.Single quotes are also allowed but it is preferable to use double quotes.If you use single quotes for quoting pattern special characters loss their meaning.

1.	`grep "sales" emp.lst`	select all lines containing the character sequence "sales"
2.	`grep "sales" emp.lst temp.lst`	output lines are prefixed by file name
3.	`grep -i "agarwal" emp.lst`	case ignore while matching pattern
4.	`grep -v "director" emp.lst`	display all lines except the lines containing director
5.	`grep -n "sales" emp.lst`	display lines numbers along with the lines
6.	`grep -c "sales" emp.lst`	count the number of lines containing the pattern sales
7.	`grep -l "manager" *.lst`	list the file names containing the pattern manager
8.	`grep -e "sales" -e "Sales" -e "SALES" emp.lst`	match multiple patterns
9.	`grep -f pattern.lst emp.lst`	patterns takesn from file,one per line
10.	`grep "^2" emp.lst`	lines begin wiht 2
11.	`grep "7...$" emp.lst`	last but 4th character is 7
12.	`grep "^[`*spacetab*`]*$"`	empty lines
13.	`grep "^$" emp.lst`	empty lines
14.	`grep "^[^2 ]" emp.lst`	lines not begin with 2
15.	`ls -l	grep "^d"`	show only directories

3.1.1 `egrep`

egrep=grep -E

1. `grep -E "sengupta|dasgupra" emp.lst`
2. `grep -E "(sen|das)gupta" emp.lst`

Table 3.1: Basic Regular Expressions (BRE)

Symbol or Expression	***Matches***
*	Zero or more occurrences of previous character
g*	Zero or more occurrences of g
.	Any single character
.*	Noting or any number of characters
$[pqr]$	A single character `p,q` or `r`
$[c1-c2]$	A single character in the range c1 to c2
$[1-9]$	A single digit from 1 to 9
`[^pqr]`	A single character which is not `p,q,` or `r`
`[^a-zA-Z]`	A single character which is not a alphabet
`^pat`	Pattern `pat` at the beginning of the line
pat $	Pattern `pat` at the end of the line
`^pat$`	Pattern `pat` as the only word in line
`^$`	Lines containing nothing

Table 3.2: Extended Regular Expression

Expression	***Significance***
ch+	Matches one or more occurrences of *ch*
ch?	Matches zero or one occurrences of *ch*
exp1|exp2	Matches *exp1* or *exp2*
(X1|X2)X3	Matches *x1x3* or *x2x3*

3.2 sed:Stream Editro

Syntax: `sed` *options 'address action' file(s)*

sed address the lines in two ways

- by line number
- by pattern

1. `sed '3q'emp.lst` quits after printing line 3
2. `sed '1,3p'emp.lst` p prints all selected lines and non selected lines. to avoid this use `-n` option.
3. `sed -n '1,3p'emp.lst` print lines 1 to 3
4. `sed -n '$p'emp.lst` print last line
5. `sed -n '1,3p`
   `7,9p`
   `$p 'emp.lst` selected multiple groups
6. `sed '3,$!p'emp.lst` action negation , dont print line 3 to last line
7. `sed -n -e '1,3p' -e '7,9p' -e '$p' emp.lst` selecting multiple groups
8. `sed -n -f instr.fil1 -f instr.fil2 emp.lst` loading instructions from file
9. `sed -n -e '1,3p' -f instr.filll emp.lst`
10. `sed -n '/director/p'emp.lst` print lines containing director
11. `sed -n '1,/director/p'emp.lst` print line 1 to line containing director
12. `sed -n '/director/,/gupta/p'emp.lst`
13. `sed -n '/director/w dlist'emp.lst` writing selected lines to file dlist
14. `sed -n '/director/w dlist`
    `/manager/w mlist`
    `/executive/w elist'emp.lst` writing to multiple files
15. `sed -n '1,500w foo1'emp.lst`

3.2.1 Inserting and Changing Lines (i,a,c)

1. `sed '1i\` *Need to use \ before [Enter] here ...*
   `>#include<std.io> \` *and here also*
   `>#include<condio.io>` *No \ in the last line of input*
   `> 'foo.c > $$ ; mv $$ foo.c` *output redirect to temporary file,rename*
   Inserted two lines in the first line of the file *foo.c*
2. `sed 'a\` *inserting after every line*
   `>` *this blank line*
   `>'emp.lst`

3.2.2 Deleting Lines (d)

1. `sed '/director/d'emp.lst` dont use `-n` option
2. `sed -n '/director/!p'emp.lst`
3. `sed '/^[␣⇥]*$/d'emp.lst` ␣ is space ⇥ is tab. delete empty lines

3.2.3 Substitution(s)

[address]s/expression1/expression2/flags

1. `sed 's/|/:/'emp.lst` substitution takes at first occurrence of | only
2. `sed 's/|/:/g'emp.lst` substitution takes at every occurrence of |
3. `sed '1,5s/|/:/g'emp.lst` substitution takes in line 1 to line 5

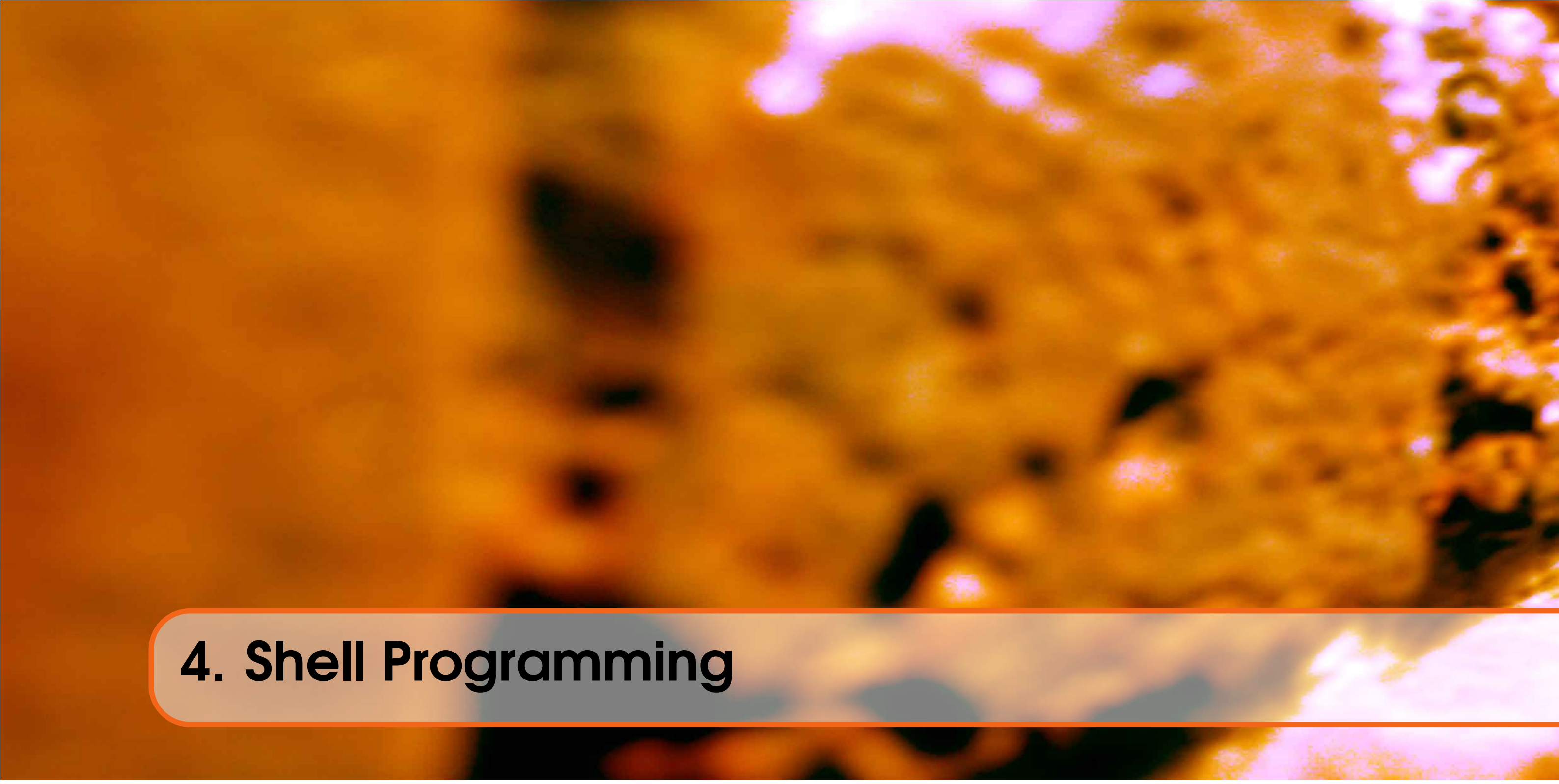
4. `sed '1,5s/director/manager /g'emp.lst` substitution takes in line 1 to line 5
5. `sed 's/^/2/g'emp.lst` 2 is placed in front of every line
6. `sed 's/$/.00/g'emp.lst` .00 is placed in end of every line
7.
   ```
   sed 's/<I>/<EM>/g
   >s/<B>/<STRONG>/g                                    multiple substitutions
   >s/<U>/<EM>/g'emp.lst
   ```
   Note When there are multiple instruction place the instructions in the file and load the file with `-f` option.

3.2.4 The Remembered Pattern(//)

1. `sed 's/director/manager/g'emp.lst`
2. `sed '/director/s//manager/g'emp.lst`
3. `sed '/director/s/director/manager/g'emp.lst`

   Note above three commands does the same work.In the second command `sed` searches for the lines containing pattern `director` and remembers the pattern `director` with // and replaces `directort` with `manager`.Third command does the same work except remembering.
4. `sed 's/|//g'emp.lst` in this command // is not remember pattern.| is removed.

   Note The significance of // depends on its position.
5. `sed '/marketing/s/director/manager/g'emp.lst`



4. Shell Programming

4.1 Shell Scripts

Comments starts with #

```
#!/bin/sh
# comments start with # symbol

echo "Today’s date is:‘date‘";
echo "This month’s calender is";
cal ‘date "+%m %Y"‘
echo "My Shell : $SHELL"
```

Listing 4.1: script.sh

4.2 read: Making Script Interactive

You can read more than one variable at a time.
`read pname file`
Note $ is not used while reading input.

```
echo "Enter the pattern to be search :\c";
read pattern;
echo "Enter the file name to search :\c";
read file;
grep "$pattern" $file;
echo "Selected recoreds shown above"
```

Listing 4.2: reading input

4.3 Command Line Arguments

Refere table 4.1.

Table 4.1: Special Parameters used by shell

Shell Parameter	Significance
$1 ,$2,..	Parameters in the sequence
$#	Number of parameters
$0	Name of the executed command(file name)
$*	list of parameters as a single string
"$@"	each quoted string is treated as separate string
$?	Exit status of previous command(0 for success,1 for failure)
$$	PID of current shell
$!	PID of last background job

```
echo "Program name:" $0;
echo "Number of paramaters:"$#
echo "Paramaters are:"$*;
grep "$1" $2;
echo "Exit status of previous commands is:" $?
```

Listing 4.3: command line arguments

4.4 Logical && and ||

cmd1* || *cmd2 *cmd2* is executed only if *cmd1* fails
cmd1* && *cmd2 *cmd2* is executed only if *cmd1* successes

```
echo "Name of the program is :$0 \n"
echo "Number of arguments is : $# \n"
echo "Arguments are : $* \n";

grep "$1" $2 || echo "pattern not fond" && exit 2

echo "This statement will not executed";
```

Listing 4.4: logical && and ||

4.5 The if Conditional

Syntax1:
if *command is successfull*
then
execute commands
else
execute commands

fi

Syntax2:
if *command is successfull*
then
execute commands

fi

Syntax3:
if *command is successfull*
then
execute commands
elseif *command is successfull*
then
execute commands
elseif..
then..
else
fi

```
if grep "$1" $2 > out 2> error
    then echo "pattern found,out put is plase in output file"
else echo "pattern not found: "
fi
```

Listing 4.5: if statement **emp4.sh**

4.6 using test and []

test and [] are used to

- compare two numbers
- compare two strings
- compare string with null
- check file attributes

4.6.1 Numerical comparison

in the following program output of the echo statement is redirected to terminal and grep output is redirected to foo file.

```
if [ $# -eq 0 ]
 then echo "usage: $0 pattern file_name" > /dev/tty
elif [ $# -lt 2 ]
    then echo "usage: $0 pattern file_name" > /dev/tty
elif grep "$1" $2
  then echo "pattern found"
  else echo "pattern not found"
  fi
```

Listing 4.6: [] example

```
$ sh emp5.sh > foo
usage: emp5.sh pattern file_name
$ sh emp5.sh sales >foo
usage: emp5.sh pattern file_name
$sh emp5.sh sales emp.lst > foo
```

Listing 4.7: [] example output

Table 4.2: Numerical comparison operaters

symbol	meaning
-gt	>
-lt	<
-eq	=
-le	<=
-ge	>=

Table 4.3: String comparison

Test	True if
s1 != s2	s1 and s2 are not equal
s1 = s2	s1 and are equal
-n *stg*	string *stg* in not null
-z *stg*	string *std* is null
stg	string std is assigned and not null
s1==s2	s1 and s2 are equal(Korn and Bash only)

4.6.2 String Comparison

AND -a OR -o are used in [] for complex comparison operations

```
if [ $# -eq 0 ]
then
       echo "Enter the pattern \n"
       read pattern
       if [ -z  $pattern ]
         then
    echo "You entered null pattern \n"
    exit 1
  fi

        echo "Enter the file name \n"
  read file
  if [ -z $file ]
   then
   echo "You entered null in the file name\n"
    exit 1
  fi

  sh emp4.sh "$pattern" $file
else

 sh emp4.sh "$1" $2

fi
```

Listing 4.8: String comparison

4.6.3 File Test

```
if [ ! -e $1 ]
```

Table 4.4: File related test

Test	True if
-f *file*	*file* exists and is a regular file
-r *file*	*file* exists and is readable
-w *file*	*file* exists and is writable
-x *file*	*file* exists and is executable
-d *file*	*file* exists and is directory
-s *file*	*file* exists and has size grater than zero
-e *file*	*file* exists (Korn and bash only)
-u *file*	*file* exists and has SUID bit set
-k *file*	*file* exists and has sticky bit set
-L *file*	*file* exists and is a symbolic link(Korn and bash only)
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)

```
then
  echo "File not exist";
elif [ ! -r $1 ]
then
  echo "echo File is not readable";
elif [ ! -w $1 ]
then
  echo "File is not writable";
else
  echo "File is both readable and writable";


fi
```

Listing 4.9: File test

4.7 The case Conditional

case can't handle relational and file tests but it can handle strings with compact code

```
echo "
Menu \n
1.ls \n
2.date\n
3.cal\n
4.who\n
5.exit \n
Enter your option\n"

read option

case $option in

1) ls -l  ;;
2) date ;;
3) cal ;;
4)who ;;
5)exit 1 ;;
```

```
*)  echo "invalid option"

esac
```

Listing 4.10: case

```
case `date | cut -d " " -f 1 ` in
Mon) echo "Today is monday" ;;
Tue ) echo "Today is tuesday" ;;
Wen) echo "Today is wednesday";;
Thu) echo "Today is thursday";;
Fri ) echo "Today is Friday";;
*) echo "Today is Holiday";;
esac
```

Listing 4.11: case with compact code

4.7.1 Matching Multiple Patterns

```
echo "do you wnat to continiue ? \n"
read answer

case $answer in

y|Y) echo "s" ;;
n|N) echo "n"
esac
```

Listing 4.12: case multiple patterns

4.7.2 Wild-Cards: case Uses Them

```
echo "do you wnat to continiue ? \n"
read answer

case $answer in

[yY][eE]*) echo "s" ;;
[nN][oO]) echo "n"
esac
```

Listing 4.13: case wild-cards

4.8 Computation

4.8.1 expr

You should provide space between operators and operands.

```
x=`wc -l  emp.lst | cut -d " " -f 1`
y=`wc -l  emp.lst| cut -d " " -f 1`
echo "$x \n"
echo "$y \n"
z=`expr $x + $y`
echo $z
expr $x + $y
expr $x - $y
expr $x * $y       #genarates syntax error Astrisk has to be escaped
expr $x \* $y
expr $x / $y
expr $x % $y
```

Listing 4.14: expr arithmetic

4.8.2 String Handling

Length of string:

$ expr "yugandharreddyakkisetty" : ’.*’ space on either side of : required

23

```
while echo "Enter your name"; do  # echo returns true always
  read name
  if [ ‘expr $name : ’.*’‘ -gt 20 ]
  then
    echo "Name lenth is grater than 20"
  else
    break;
  fi
done
```

Listing 4.15: `expr` Finding length

4.9 `wile` Looping

syntax:

`while` *condition is true*

`do`

commands

`done`

```
#!/bin/sh
# emp5.sh: Shows use of the while loop
#
answer=y          # Must set it to y first to enter the loop
while [ "$answer" = "y" ]      # The control command
do
    echo "Enter the code and description: \c" >/dev/tty
    read code description      # Read both together
    echo "$code|$description" >> newlist     # Append a line to newlist
    echo "Enter any more (y/n)? \c" >/dev/tty
    read anymore
    case $anymore in
         y*|Y*) answer=y ;;    # Also accepts yes, YES etc.
         n*|N*) answer=n ;;    # Also accepts no, NO etc.
             *) answer=y ;;    # Any other reply means y
    esac
done
```

Listing 4.16: `while` looping

You can use output redirection operator at the end of loop

`done > newlist`

```
#!/bin/sh
# monitfile.sh: Waits for a file to be created
#
while [ ! -r emp.lst ]      # While the file invoice.lst can't be read
do
   sleep 60                       # sleep for 60 seconds
done
```

```
sh emp6.sh                              # Execute this program after exiting loop
```

Listing 4.17: while looping waiting for file

4.9.1 Infinate loop

while true;
do
df -h
sleep 300
done &

loops run in background.Print output for every five minutes

4.10 for Loop

```
for pattern in $* ;  do
grep "$pattern" emp.lst || echo "$pattern not found \n"

done
```

Listing 4.18: for looping

```
for file in `ls *.sh` ;  do
cat $file >>total.sh

done
```

Listing 4.19: for looping

4.11 set and shift

```
$ set 1202 1203 1204 1205
$ echo "\$1 is $1 \$2 is $2"
$1 is 1202 $2 is 1203
$ echo "The $# arguments are $*"
The 4 arguments are 1202 1203 1204 1205
$ shift
$ echo "The $# arguments are $*"
The 3 arguments are 1203 1204 1205
$ shift
$ echo $1
1204

$set `date`
$echo $*
Mon Aug 22 13:46:28 IST 2016
$echo "Today date is $2,$3,$6"
Today date is Aug,22,2016
```

Listing 4.20: set and shift

5. System Administration

5.1 Managing Disk Space

5.1.1 df : Reporting Free Space

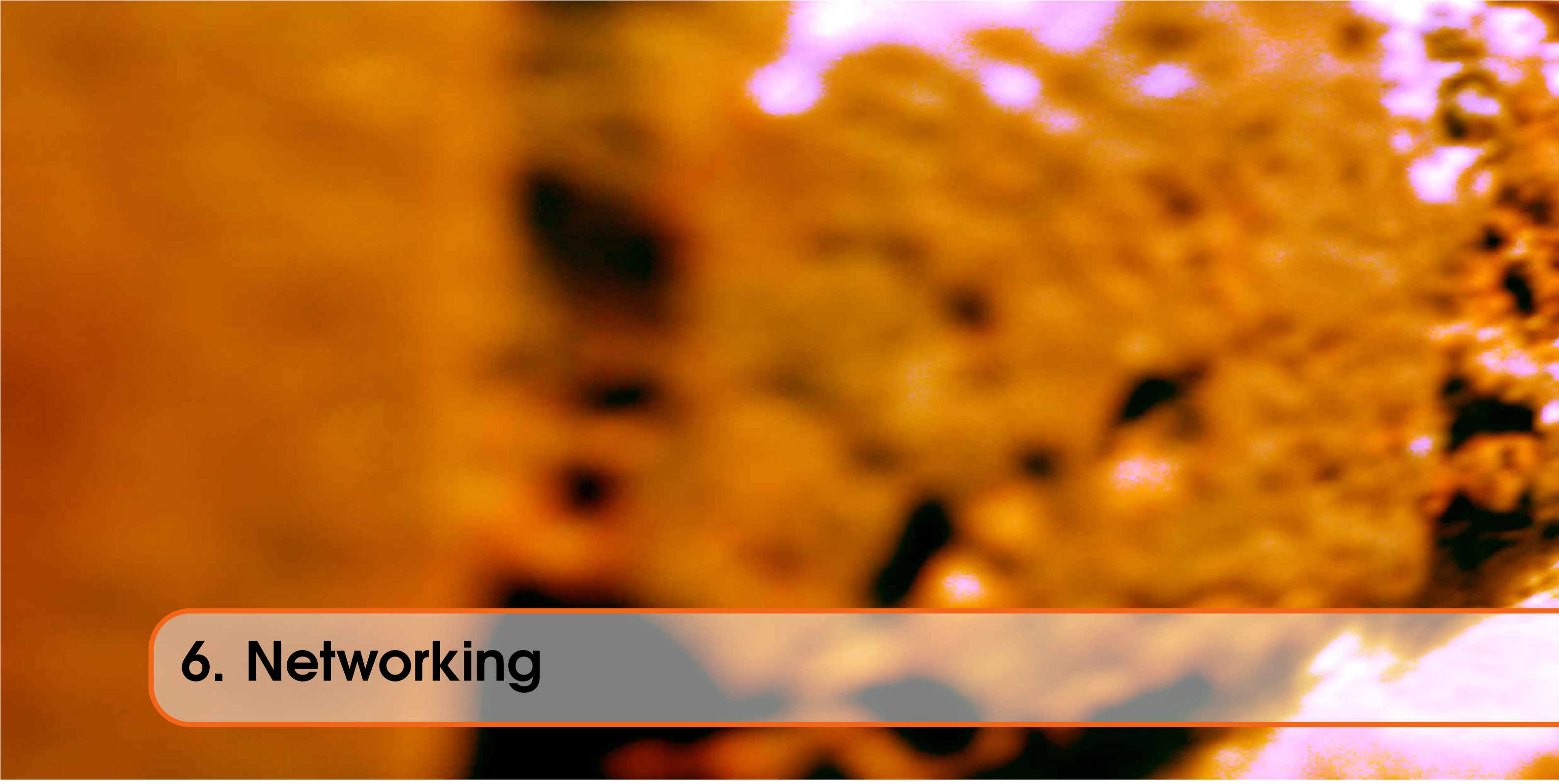
1.	`df`	amount of free space available on each file system
2.	`df -k`	free space reported in KBs
3.	`df -k / /home`	report fee space of /,/home file systems in KBs
4.	`df -h`	human readable format
5.	`df -h / /home`	

`df` is always used with file systems.To disk usage of directory tree you should use `du`

5.1.2 du: disk usage

By default `du` find out the consumption of a specific directory tree by recursive examination of the directory tree and finally report summary.

1.	`du /home/yugandhar/shell`	
2.	`du -h /home/yugandhar/shell`	human readable format
3.	`du -s /home/*`	space consumed by the users
4.	`du -sk /home/*`	space consumed by the users in KBs
5.	`du -s /home/yugandhar/shell`	
6.	`du -a /home/yugandhar/shell`	report space consumed by files also



6. Networking

6.1 Basic Networking Commands

6.1.1 telnet

1. `telnet` : Command is used to remote access.

   $ `telnet 10.66.59.36` You can use name of the server to connect

   ```
    Trying 10.66.59.36 ...
   Connected to localhost.
   Escape character is ’ˆ]’.
   Ubuntu 14.04.3 LTS
   yugandharreddy login: rstudio
   Password:
   ```

   $ `rstudio@yugandharreddy:~$ pwd` now you are in home directory

   \home\rstudio

   Now you can use all linux commands

6.1.2 ftp

`ftp` is used to transfer the files.once you connected to the server with `ftp` protocol you can use file system commands `pwd`,`cd`,`rm`,`mkdir`.By default these commands applies to remote system.To use these on local system use `!` in front of them.

1. $ `ftp` 10.66.59.36

   ```
   Connected to yugandharreddy.
   220 ProFTPD 1.3.4c Server (ProFTPD) (::ffff:127.0.0.1)
   Name (yugandharreddy:yugandhar): yugandhar
   331 Password required for yugandhar
   Password:
   230 User yugandhar logged in
   ```

Remote system type is UNIX.
Using binary mode to transfer files.
ftp>

2.	ftp> `pwd`	display remote system pwd
3.	ftp> `!pwd`	display local system pwd
4.	ftp> `cd ajax`	
5.	ftp> `get ajax.pdf`	download ajax.pdf to local system pwd
6.	ftp> `get ajax.pdf notes.pdf`	file name changed to notes.pdf in local system
7.	ftp> `mget ajax.pdf ajax.db ajax.php`	download multiple files from remote system
8.	ftp> `put php.pdf`	upload the php.pdf to remote system
9.	ftp> `put php.pdf wt.pdf`	change file name to wt.pdf in remote system
10.	ftp> `mput php.pdf ajax.pdf jsp.pdf asp.pdf`	upload multiple files to remote system
11.	ftp> `bye`	Terminate ftp session

Part Two

7. awk-Advance Filter

7.1 Simple awk Filter

Syntax:
awk *options 'selection_criteria {action}' file(s)*

Note:Selection_criteria or action any one of them can be absent but not both

1. `$ awk '/director/{print}'emp.lst`
   Prints all the lines containing the word director

   Following three commands does same task
   ```
   awk '/director/'emp.lst #### default action is print
   awk '/director/{print}'emp.lst ####
   awk '/director/{print $0}'emp.lst    #### $0 represents complete line
   ```

2. `$awk -F "|" '/sa[kx]s*ena/ {print}'emp.lst`
   For pattern matching **awk** uses **sed**-style regular expressions.You can use multiple patterns but each pattern should be separated by |

7.2 Spliting Lines into Fields

awk uses continuous spaces and tabs as a *single* delimiter. You can specify your own delimiter using **-F** option

1. `awk -F "|" '/sales/{print $2,$3,$4,$6}'emp.lst`

```
g.m. sales 6000
director sales 6700
manager sales 5600
manager sales 5000
```

2. `awk -F "|" 'NR==3,NR==6 {print $3}'emp.lst`
   NR is line number,it prints field 3 from every record from record number 3 to 6

7.2.1 Redirecting output

1. `awk -F "|" 'NR==3,NR==6 {print $3>"awk2.txt"}'emp.lst`

```
d.g.m.
director
chairman
director
```

2. `awk -F "|" 'NR==3,NR==6 {print $3 | "sort"}'emp.lst`

```
chairman
d.g.m.
director
director
```

7.3 Comparison Operators

1. **awk -F "|" '$3=="director"||$3=="chairman" {print $1}'emp.lst**
   || **or** operator ,&& **and** operator

   ~ and **!~** are used for regular expression matching and not matching
2. `awk -F "|" '$3~/director|chairman/'emp.lst`
   print all lines containing the word director or chairman

3. `awk -F "|" '$3!~/director|chairman/'emp.lst`
   print all lines except lines containg director

4. `awk -F "|" '$4~/[sS]ales/'emp.lst`

   Number comparision

5. `awk -F "|" '$6>7500 {print $1}'emp.lst`
6. `awk -F "|" '$6>8000 || $5~/45$/{print $1}'emp.lst`

7.4 Number Processing

1. `awk -F "|" '$3~/director/{print $2,$3,$6,$6*0.15,$6*0.12}'emp.lst`

7.4.1 Variables

1. `awk -F "|" '$3~/director/{kount++;print kount,$1}'emp.lst`

7.5 -f Option

1. `awk -F "|" -f empawk.awk emp.lst`

```
$3=="director"{
  print ++kount,$1;
}
```

Listing 7.1: awk program in file

Table 7.1: Built in awk variables

Variable	Function
NR	Line number
FS	Input field separater
OFS	Output field separater
FILENAME	file name where matches occures (multiple input files)
ARGC	No.of Arguments
ARGV	List of arguments

7.5.1 BEGIN and END Sections

1. `awk -F "|" -f empawk2.awk emp.lst`

```
BEGIN{

  print "\t Employee Details in the Sales in department\n"
}

$4~/[sS]ales/ {
  kount++;
  printf "%d %s \n", kount,$0;
  tot+=$6;
}

END{
  printf "\t Average of salaries in the sales department %d\n",tot/kount ;
}
```

Listing 7.2: awk program in file

7.5.2 Bulit-in variables

1. `awk -f empawk3.awk emp.lst`

```
BEGIN{
  FS="|";
  OFS="−%−";
  print "\t Employee Details in the Sales in department\n"
}

$4~/[sS]ales/ {
  kount++;
  print kount,$0;
  tot+=$6;
}

END{
  printf "\t Average of salaries in the sales department %d\n",tot/kount ;
}
```

Listing 7.3: awk program in file

7.6 Arrays

1. `awk -f empawk4.awk emp.lst`

```
BEGIN {
   FS = "|"
   printf "%46s\n", "Basic     Da    Hra  Gross"
} /sales|marketing/ {
   # Calculate the da, hra and the gross pay
   da = 0.25*$6 ; hra = 0.50*$6 ; gp = $6+hra+da

   # Store the aggregates in separate arrays
   tot[1] += $6 ; tot[2] += da ; tot[3] += hra ; tot[4] += gp
   kount++
}
END { # Print the averages
   printf "\t      Average    %5d %5d %5d %5d\n", \
   tot[1]/kount, tot[2]/kount, tot[3]/kount, tot[4]/kount
}
```

Listing 7.4: awk program in file

7.6.1 Associative Arrays

1. `awk -f empawk5.awk`

```
BEGIN{
  directions["N"]="North";
  directions["S"]="South";
  directions["E"]="East";
  directions["W"]="West";
  printf "%s \n", directions["N"];
  printf "%s \n", directions["E"];
  printf "%s \n", directions["W"];
  printf "%s \n", directions["S"];
}
```

Listing 7.5: awk program in file

`ENVIRON[]`: Array

some times you want to know the user or home directory of the user how currently running the programm

1. `awk -f empawk6.awk`

```
BEGIN{
 print ENVIRON["HOME"] "\n";
 print ENVIRON["PATH"] "\n";
}
```

Listing 7.6: awk program in file

7.7 Function

1. `awk -f empawk7.awk`

Table 7.2: Built in functions in **awk**

Function	**Description**
int(x)	Returns the integer value of x
sqrt(x)	Returns the squre root of x
length	Returns the length of the current line
length(x)	Returns the length of x
substr(stg,m,n)	Returns the sub string of length n starting from the index m in string stg
index(s1,s2)	Returns the index of s2 in s1
split(stg,arr,ch)	Split the string stg into arr array using ch as delimiter and Returns the number of fields
system("cmd")	Executes the cmd system command and returns it's exit status

```
BEGIN{
   FS="|";
 system("tput clear");
 system("date");
}
 substr($5,7,2) > 45 && substr($5,7,2) < 52 {
 print ; # print current line
 print length; # print length of current line
 split($5,arr,"/"); #date of birth in DDMMYYYY format
 print arr[1]arr[2]"19"arr[3]

}

END{
    print "END";
   }
```

Listing 7.7: awk program in file

7.8 Control Flow

this control flow should be use in BEGIN,END and action part only. not in selection_criteria

7.8.1 `if`-statement

C language syntax is used
$ awk -F "|" {if (NR>=3 && NR<=6) print}
$ awk -F "|" {if ($3~/^g.m/) print}

7.8.2 `for` -loop

awk used two flavours in for loop
1. C- style for loop, Syntax is same as in th C
2. for - in loot
Syntax
for (*key* in *arr*)
{
commands

```
}
```

1. `awk -f empawk8.awk`

```
BEGIN{
  FS="|";
  }
{kount[$3]++}
END{
  for (key in kount)
  print key"="kount[key] ;
  }
```

Listing 7.8: frequency count

2. `awk -f empawk9.awk`

```
BEGIN{
  for(key in ENVIRON)
  {
    print key"="ENVIRON[key];
  }
}
```

Listing 7.9: print all environment variable

7.8.3 `while`

same rules as in C

8. PERL

8.1 Introduction

8.1.1 Running Perl

The are following are the different ways to start Perl.

1. Interactive Interpreter
   `$perl -e \<perl code>\`
2. Script from the Command-line
   `$perl script.pl`

With the following command you can test wehter the `perl` is in you PATH.

```
$ perl -e 'print("Hello World");'
```

8.1.2 Comments in Perl

- Single line comments starts with #
- Multi line comments can be written in the following way

```
=begin comment
you can write comments
in the multi line
=cut
```

8.1.3 `print` Statement in Perl

Single quotes ignores meaning of special characters.

```
• print "Hello
  World\n"                                    Hello world printed in two lines
•  $a=10;
  print "value of a=$a";                                         value of a=10
  print 'value of a=$a';                                         value of a=$a
```

8.1.4 HERE Document

Double quotes in HERE documents evaluate the variables inside it,but single quotes will not evaluate them.

```
$var=«"EOF"
This is some text
This is next line variables in side it will evaluate
EOF

    $var=«'EOF'
This is some text
This is next line variables in side it will evaluate
EOF
```

8.1.5 Escaping Character(\)

```
 $result="this is \"number\"";
print "$result\n";                                      this is "number"
print "\$result\n"                                              $result
```

8.1.6 The chop Function

Following program reads input from slandered input.*chop()* is used to remove the last character.In this case next line character is removed.

```
print("Enter your name \n");
chop($name=<STDIN>);
print "$name";
```

Listing 8.1: Input read from key board

```
print "Enter your name\n";
chop($name=<STDIN>);
if($name ne ""){
 print "$name is entered\n";}
else{
 print "Enterred null\n";}
```

Listing 8.2: If statement

8.2 Variables and Operators

Perl identifiers are case sensitive.perl variable are start with $,@ and %.`$power` and `$Power` are two different variables.There are three data types in the variables.

Scalar start with $
Array start with @,index start with 0
Hashes start with %,index is string
You can use same name for scalar,array and hashe.

`perl` variables have no type and need no initialization.

- String automatically converted into numeric when needed.
- If a variable is undefined it is assumed to be null string and a null string is numerical zero.The following command pints 1.
  `$ perl -e '$x++ ; print("$ \n");'`
- If the first character of a string is not numeric,the entire string becomes numerically equivalent to zero

some examples on variables

`$x="X";$x++` This becomes Y

`$x="Xy1";$x++`	This becomes Xy2
`$x="XY";$x++`	This becomes XZ
`$x="X";$x++`	This becomes Y
`$x="yugandhar reddy";$y="\U$x\E"`	This becomes YUGANDHAR REDDY
`$x="yugandhar reddy";$y="\u$x\E"`	This becomes Yugandhar reddy

8.2.1 Special Literals

`print "File name is:_FILE_\n"; prints file name print "Line number:_LINE_\n"; prints line number print "Package:_PACKAGE_\n"; dont use these in same line` Conditional operator is used in the following program.

```
print "Enter year\n";
chop($year=<STDIN>);
$feb=$year%4==0?29:28;
print "$feb\n";
```

Listing 8.3: conditional statement

Concatenation Operator . and x

- `$ perl -e '$x=maruti;$y=".com";print($x.$y."\n") '`
- `$ perl -e 'print "*" x 40'` * printed 40 times

8.2.2 $_ Current Line,$. Current Line number,Range operator(..)

Following program takes input from the command line through the operator <>.Number of iterations of the while loop is equal to number of lines in the file passed in the command line.$_ represents current line.`$.` stores current line number.By default regular expression in the if statement is matched against the current line.`if(/gupta/)` is equal to `if($_=~/gupta/)`

```
while(<>)
{
  if(/gupta/)
  {
  print $_;
}

}
```

Listing 8.4: Search for string

```
2365|barun sengupta|director|personnel|11/05/47|7800
5423|n.k. gupta|chairman|admin|30/08/56|5400
1265|s.n. dasgupta|manager|sales|12/09/63|5600
```

Listing 8.5: out for above code

```
while(<>)
{
   #actually ($_=<>)
  if(/gupta/)
  {
    print ($. .":".$_);
  }
}
```

Listing 8.6: Search for strin with line number

```
4:2365|barun  sengupta|director|personnel|11/05/47|7800
5:5423|n.k.  gupta|chairman|admin|30/08/56|5400
8:1265|s.n.  dasgupta|manager|sales|12/09/63|5600
```

Listing 8.7: output for the above program

```
  $ perl -e 'if (3..8)  print $_'
$ perl -e 'print if(3..8)'
```

above both commands does same thing.default comparison is done with line number that is $. So line 3 to 8 are printed.

8.3 Arrays

In case of arrays, index starts from zero in perl like C.

```
@var=(1,2,3,4);
print @var;                                                        1234
@list=(10..15);
print @list;                                               101112131415
```

Special variable $(

```
 print "$[";                           print the default start index of array
$[=1;                                       setting array start index to 1
```

Merging Arrays

```
 @var=(1,2,3,(4,7,6));
@eve=(2,4,6,8);
@odd=(1,3,5,7,9);
@numbers=(@eve,@odd);
```

Selecting Array Elements

```
 @var=(1..10)[4,5];  4,5 indexed elements of the list are selected in to the @var list
@var=(20..30);
@list=@var[0..9]                             first 10 elements are selected into list
```

First line of the code stores total file in the array `line`.Each element of the `line` is one record in the file supplied through the command line.In the statement `$size=@line` number of elements in the array `@line` is assigned to scalar `$size`

```
@line=<>;
$index=0;
$size=@line;
print $size;
while($index<$size)
{
  #print "$index \n";
  if($line[$index]=~/gupta/)
  {
    print $line[$index];
  }
  $index++;
}

#print @line;
```

Listing 8.8: Read file into array

```
152365|barun sengupta|director|personnel|11/05/47|7800
5423|n.k. gupta|chairman|admin|30/08/56|5400
1265|s.n. dasgupta|manager|sales|12/09/63|5600
```

Listing 8.9: Read file into array output

8.3.1 Array Handling Functions

splice() can do more than other functions.It takes up to four arguments to add or remove elements at any location or array.First argument is list,second argument is offset from where insertion and removal should begin,third argument represents number of elements to remove.If it is 0,elements have to be added.The new replaced list is specified by the fourth argument.

```
@list=(1,2,3,4,5);
#array size
$size=@list;
print $size ."\n";
# remove element in front

shift(@list);
print "@list" . "\n";

#remove element at back

pop(@list);
print "@list" . "\n";

# insert in front

unshift(@list,1);
print "@list" . "\n";

#insert in end
push(@list,5);
print "@list" . "\n";

#splice can do all
# splice uses four arguments,1-list,2.from edit start,3.#elements to remove,add if 0,
#4.elements to insert

splice(@list,2,0,6..8);
print "@list" . "\n";
splice(@list,2,3);
print "@list" . "\n";
```

Listing 8.10: Array related functions

```
5
2 3 4 5
2 3 4
1 2 3 4
1 2 3 4 5
1 2 6 7 8 3 4 5
1 2 3 4 5
```

Listing 8.11: Array related functions output

```
@month=qw/jan feb mar apr may jun jul aug sep oct nov dec/;
#print array
print @month ;
print "\n";
```

```
#print last index
print $#month ."\n";
#print  fixing last index
$#month=5;
print @month;
print "\n";
```

Listing 8.12: Array last index

```
janfebmaraprmayjunjulaugsepoctnovdec
11
janfebmaraprmayjun
```

Listing 8.13: Array last index output

`sort()`

This function is used to sort the list.Following script shows how to use this sort function.
`@list=(85,74,96,41,52,35); @list=sort(@list);`

`foreach`:Printing array

Following program takes command line arguments.Those arguments are stored in @ARGV[]
`$perl foreach.pl 10 20 30 40`

```
# @ARGV[] stores command line arguments

foreach $var (@ARGV)
{
  print "Squre root of $var is ". sqrt($var) ."\n";
}
```

Listing 8.14: Foreach loop

```
Squre root of 10 is 3.16227766016838
Squre root of 20 is 4.47213595499958
Squre root of 30 is 5.47722557505166
Squre root of 40 is 6.32455532033676
```

Listing 8.15: Foreach loop output

split()

split() splits the string into list or scalars.Following two programs shows how this function works. Following program takes a string
`split() splits the string into list/scalars`
Syntax: `@list=split(/sep/,strg);`
`or ($var1,$var2,...,$varn)=split(/sep/,strg);`

```
print "Enter a strings\n";
chop($numbers=<STDIN>);
@list=split(/ /,$numbers);
print $list[0] ."\n";
print $list[$list -1] ."\n";
```

Listing 8.16: Split string

Output for the above program

```
Enter a strings
this is yugandhar reddy
this
reddy
```

```
@line=<>;
$size=@line;
while($index<$size)
{    #chop($line[$index]);
   if($line[$index]=~/gupta/)
   {
      print $line[$index] . "\n";
      @record=split (/\|/ , $line[$index]);
      print $record[0] .":" .$record[5] . "\n";
      #print @record ;
   }
   $index++;
}
```

Listing 8.17: Split file

```
2365|barun sengupta|director|personnel|11/05/47|7800

2365:7800

5423|n.k. gupta|chairman|admin|30/08/56|5400

5423:5400

1265|s.n. dasgupta|manager|sales|12/09/63|5600

1265:5600
```

Listing 8.18: Split file output

join()

`join()` is used to join the strings in to list

Syntax:

`$strg=join(separater,@list1,@list2,...,@listn)`

```
while(<>)
{
   @record=split(/\|/);
   ($day,$month,$year)=split(/\//,$record[4]);
  #print $day . $month . $year;
   $year="19".$year;
   $record[4]=join("\/",$day,$month,$year);
   $record=join("\|",@record);
   print $record;



}
```

Listing 8.19: join

```
2233|a.k. shukla|g.m.|sales|12/12/1952|6000
9876|jai sharma|director|production|12/03/1950|7000
5678|sumit chakrobarty|d.g.m.|marketing|19/04/1943|6000
```

Listing 8.20: join output: perl join.pl emp.lst | head -n 3

8.3.2 `grep`:Array Search

searching array with grep

```
@line=<>;
@found_array=grep(/^2/,@line);
print @found_array;
```

Listing 8.21: search array with grep

```
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
```

Listing 8.22: search array with grep output

```
@list=<>;
for(;;)
{
  print "Enter regural expression to search\n";
  chop($regexp=<STDIN>);
  die("Good Bye") if($regexp eq "exit");

  if($regexp eq "")
  {
    print "Enter the regular expression\n";
    next; #continue next itaration
  }
  @found=grep(/$regexp/,@list);#it returns more than one record
  if($#found==-1)
  {
    print "Expression not found\n";
    next;
  }

  for($i=0; $i<=$#found; $i++)
  {
    print $found[$i];
    @record=split(/\|/,$found[$i]);
    print $record[0];
    print "\n";
  }

}
```

Listing 8.23: search array with grep

```
Enter regural expression to search
^2.
2233|a.k. shukla|g.m.|sales|12/12/52|6000
2233
2365|barun sengupta|director|personnel|11/05/47|7800
2365
2476|anil aggarwal|manager|sales|01/05/59|5000
2476
2345|j.b. saxena|g.m.|marketing|12/03/45|8000
2345
Enter regural expression to search
8...$
6521|lalit chowdury|director|marketing|26/09/45|8200
6521
2345|j.b. saxena|g.m.|marketing|12/03/45|8000
2345
Enter regural expression to search
```

```
exit
```

Listing 8.24: search array with grep output

8.4 Hashes(Associative Arrays)

Arrays with indexes as strings are called associative arrays. Associative arrays are start with %,Following program shows how to create hashes, how to access the elements form hashes,how to get the list of keys and elements form the hashes.

```
%family=("l","lokanath reddy","k","kamalakshi","y","yugandhar reddy","s","sree lakshmi","c","chandra sekhar reddy");
@index=keys(%family); #list of subscripts
@value=values(%family); #list of values
print "enter one key from the following\n";
print @index;
print "\n";

chop($subscript=<STDIN>);
print $family{$subscript};
print "\n";
```

Listing 8.25: associative arrays

output for the above program

```
enter one key from the following
lysck
l
lokanath reddy
```

8.4.1 Finding length

`keys()` will return list of keys of a associative array into a list. If we find the size of the list returned by `keys()` that means we find length of the associative array.

```
@list=keys(%family);
$size=@list
```

8.4.2 Insert,delete and exists

You can add the new element in to associative array ,delete a element form it. Following script shows how to do so.

```
%family{"r"}="Red";                                              insert
if(exist($family{"r"}))
{
delete $family{"r"}
}
```

8.4.3 *Application*:Frequency count

```
while(<>)
{
        @record=split (/\|/); # default current line;split result stored in $_[]
  $dept=$record[3];
  $deptlist{$dept}+=1;
}

```

```
foreach $dept (keys(%deptlist))
{
  print "$dept=$deptlist{$dept}\n";
}
```

Listing 8.26: counting occurrence

```
marketing=4
accounts=2
sales=4
personnel=2
admin=1
production=2
```

Listing 8.27: counting occurrence output

8.5 Control Statements

8.6 Date and Time

We can get the local time of the system by `localtime()`. `localtime()` returns the current time of the system.`time()` returns number of second that have elapsed since the given date in UNIX is Jan 1,1970.The following program shows various manipulation on date and time.

```
$date=localtime();
@date=split(/ /,$date);
print $date[3];
print "\n";
$epoch=time();
print $epoch;
print "\n";
$epoch=$epoch-24*60*60;
$yesday=localtime($epoch);
print $yesday ;
```

Listing 8.28: date and time

8.7 Subroutines

Define and Call a Subroutine

The general form of a subroutine definition in Perl programming language is as follows

```
sub subroutine_name{
body of the subroutine
}
```

The typical way of calling that Perl subroutine is as follows

```
subroutine_name( list of arguments );
```

Following program shows example of subroutine.

```
#!/usr/bin/perl

# Function definition
sub Hello{
   print "Hello, World!\n";
}

# Function call
Hello();
```

Listing 8.29: subroutines

8.8 Passing Arguments to a Subroutine

You can pass various arguments to a subroutine like you do in any other programming language and they can be acessed inside the function using the special array @_. Thus the first argument to the function is in $_[0], the second is in $_[1], and so on.

You can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities. So we will use references to pass any array or hash.

```
#!/usr/bin/perl

# Function definition
sub Average{
   # get total number of arguments passed.
   $n = scalar(@_);
   # $n=@_; is equal to above statement
   $sum = 0;

   foreach $item (@_){
      $sum += $item;
   }
   $average = $sum / $n;

   print "Average for the given numbers : $average\n";
}

# Function call
Average(10, 20, 30);
```

Listing 8.30: passing arguments

8.8.1 Passing Lists to Subroutines

Because the @_ variable is an array, it can be used to supply lists to a subroutine. However, because of the way in which Perl accepts and parses lists and arrays, it can be difficult to extract the individual elements from @_. If you have to pass a list along with other scalar arguments, then make list as the last argument as shown below.

```
#!/usr/bin/perl

# Function definition
sub PrintList{
    @list = @_;
   print "Given list is @list\n";
}
$a = 10;
@b = (1, 2, 3, 4);

# Function call with list parameter
PrintList($a, @b);
```

Listing 8.31: passing list as arguments

8.8.2 Passing Hashes to Subroutines

When you supply a hash to a subroutine or operator that accepts a list, then hash is automatically translated into a list of key/value pairs.

```
#!/usr/bin/perl

# Function definition
sub PrintList{
    @list = @_;
```

```
   print "Given list is @list\n";
}
$a = 10;
@b = (1, 2, 3, 4);

# Function call with list parameter
PrintList($a, @b);
```

Listing 8.32: passing hashes as arguments

8.8.3 Returning Value from a Subroutine

You can return a value from subroutine like you do in any other programming language. If you are not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value.

You can return arrays and hashes from the subroutine like any scalar but returning more than one array or hash normally causes them to lose their separate identities. So we will use references (explained in the next section) to return any array or hash from a function.

```
#!/usr/bin/perl

# Function definition
sub Average{
   # get total number of arguments passed.
   $n = scalar(@_);
   $sum = 0;

   foreach $item (@_){
      $sum += $item;
   }
   $average = $sum / $n;

   return $average;
}

# Function call
$num = Average(10, 20, 30);
print "Average for the given numbers : $num\n";
```

Listing 8.33: Returning value from subroutine

8.8.4 Private Variables in a Subroutine

By default, all variables in Perl are global variables, which means they can be accessed from anywhere in the program. But you can create private variables called lexical variables at any time with the **my** operator.

The **my** operator confines a variable to a particular region of code in which it can be used and accessed. Outside that region, this variable cannot be used or accessed. This region is called its scope. A lexical scope is usually a block of code with a set of braces around it, such as those defining the body of the subroutine or those marking the code blocks of *if, while, for, foreach, and eval* statements.

Following is an example showing you how to define a single or multiple private variables using my operator

```
sub somefunc {
my $variable; # $variable is invisible outside somefunc()
my ($another, @an_array, %a_hash); # declaring many variables at once
}
```

```
#!/usr/bin/perl

```

```
# Global variable
$string = "Hello, World!";

# Function definition
sub PrintHello{
   # Private variable for PrintHello function
   my $string;
   $string = "Hello, Perl!";
   print "Inside the function $string\n";
}
# Function call
PrintHello();
print "Outside the function $string\n";
```

Listing 8.34: my keyword

8.8.5 Temporary Values via local()

The **local** is mostly used when the current value of a variable must be visible to called subroutines. A local just gives temporary values to global (meaning package) variables. This is known as dynamic scoping. Lexical scoping is done with my, which works more like C's auto declarations.

If more than one variable or expression is given to local, they must be placed in parentheses. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or eval.

```
#!/usr/bin/perl

# Global variable
$string = "Hello, World!";

sub PrintHello{
   # Private variable for PrintHello function
   local $string;
   $string = "Hello, Perl!";
   PrintMe();
   print "Inside the function PrintHello $string\n";
}
sub PrintMe{
   print "Inside the function PrintMe $string\n";
}

sub PrintMore{
print "Inside the function PrintMore $string\n";
}
# Function call
PrintHello();
PrintMore();
print "Outside the function $string\n";
```

Listing 8.35: local keyword

8.8.6 State Variables via state():(static in C)

There are another type of lexical variables, which are similar to private variables but they maintain their state and they do not get reinitialized upon multiple calls of the subroutines. These variables are defined using the**state** operator and available starting from Perl 5.9.4.

```
#!/usr/bin/perl

use feature 'state';
```

```

sub PrintCount{
   state $count = 0; # initial value

   print "Value of counter is $count\n";
   $count++;
}

for (1..5){
   PrintCount();
}
```

Listing 8.36: state keyword

8.8.7 Subroutine Call Context

The context of a subroutine or statement is defined as the type of return value that is expected. This allows you to use a single function that returns different values based on what the user is expecting to receive. For example, the following localtime() returns a string when it is called in scalar context, but it returns a list when it is called in list context.

```
my $datestring = localtime( time );
```

In this example, the value of $timestr is now a string made up of the current date and time, for example, Thu Nov 30 15:21:33 2000.

```
($sec,$min,$hour,$mday,$mon, $year,$wday,$yday,$isdst) = localtime(time);
```

Now the individual variables contain the corresponding values returned by localtime() subroutine.

8.9 References

A Perl reference is a scalar data type that holds the location of another value which could be scalar, arrays, or hashes. Because of its scalar nature, a reference can be used anywhere, a scalar can be used.

Create References

It is easy to create a reference for any variable, subroutine or value by prefixing it with a backslash as follows

```
$scalarref = \$foo;
$arrayref = \@ARGV;
$hashref = \%ENV;
$coderef = \&handler;
$globref = \*foo;
```

You cannot create a reference on an I/O handle (filehandle or dirhandle) using the backslash operator but a reference to an anonymous array can be created using the square brackets as follows

```
$arrayref = [1, 2, ['a', 'b', 'c']];
```

Similar way you can create a reference to an anonymous hash using the curly brackets as follows

```
$hashref = { 'Adam'=> 'Eve', 'Clyde'=> 'Bonnie'};
```

A reference to an anonymous subroutine can be created by using sub without a sub name as follows

```
$coderef = sub { print "Boink!\n"; }
```

8.9.1 Dereferencing

Dereferencing returns the value from a reference point to the location. To dereference a reference simply use $, @ or % as prefix of the reference variable depending on whether the reference is pointing to a scalar, array, or hash. Following is the example to explain the concept

```
#!/usr/bin/perl

$var = 10;

# Now $r has reference to $var scalar.
$r = \$var;

# Print value available at the location stored in $r.
print "Value of $var is : ", $$r, "\n";

@var = (1, 2, 3);
# Now $r has reference to @var array.
$r = \@var;
# Print values available at the location stored in $r.
print "Value of @var is : ",  @$r, "\n";

%var = ('key1' => 10, 'key2' => 20);
# Now $r has reference to %var hash.
$r = \%var;
# Print values available at the location stored in $r.
print "Value of %var is : ", %$r, "\n";
```

Listing 8.37: dereferencing

If you are not sure about a variable type, then its easy to know its type using **ref**, which returns one of the following strings if its argument is a reference. Otherwise, it returns false

```
#!/usr/bin/perl

$var = 10;
$r = \$var;
print "Reference type in r : ", ref($r), "\n";

@var = (1, 2, 3);
$r = \@var;
print "Reference type in r : ", ref($r), "\n";

%var = ('key1' => 10, 'key2' => 20);
$r = \%var;
print "Reference type in r : ", ref($r), "\n";
```

Listing 8.38: ref-knowing reference type

When above program is executed, it produces the following result

```
Reference type in r :  SCALAR
Reference type in r :  ARRAY
Reference type in r :  HASH
```

8.9.2 Circular References

A circular reference occurs when two references contain a reference to each other. You have to be careful while creating references otherwise a circular reference can lead to memory leaks. Following is an example

```perl
!/usr/bin/perl

 my $foo = 100;
 $foo = \$foo;

 print "Value of foo is : ", $$foo, "\n";
```

Listing 8.39: Circular Reference

When above program is executed, it produces the following result

```
Value of foo is :  REF(0x9aae38)
```

8.9.3 References to Functions

This might happen if you need to create a signal handler so you can produce a reference to a function by preceding that function name with \& and to dereference that reference you simply need to prefix reference variable using ampersand &. Following is an example

```perl
#!/usr/bin/perl

# Function definition
sub PrintHash{
   my (%hash) = @_;

   foreach $item (%hash){
      print "Item : $item\n";
   }
}
%hash = ('name' => 'Tom', 'age' => 19);

# Create a reference to above function.
$cref = \&PrintHash;

# Function call using reference.
&$cref(%hash);
```

Listing 8.40: Subroutine Reference

8.10 File I\O

The basics of handling files are simple: you associate a filehandle with an external entity (usually a file) and then use a variety of operators and functions within Perl to read and update the data stored within the data stream associated with the filehandle.

A filehandle is a named internal Perl structure that associates a physical file with a name. All filehandles are capable of read/write access, so you can read from and update any file or device associated with a filehandle. However, when you associate a filehandle, you can specify the mode in which the filehandle is opened.

Three basic file handles are - STDIN, STDOUT, and STDERR, which represent standard input, standard output and standard error devices respectively.

8.10.1 Open Function

Following is the syntax to open emp.lst in read-only mode. Here less than < sign indicates that file has to be opend in read-only mode.

`open(DATA, "<emp.lst");`

Here DATA is the file handle which will be used to read the file. Here is the example which will open a file and will print its content over the screen.

Table 8.1: The possible values of different modes

Entities	Definition
< or r	Read Only Access
> or w	Creates, Writes, and Truncates
» or a	Writes, Appends, and Creates
+< or r+	Reads and Writes
+> or w+	Reads, Writes, Creates, and Truncates
+» or a+	Reads, Writes, Appends, and Creates

```perl
#!/usr/bin/perl

open(DATA, "<emp.lst") or die "Couldn’t open file emp.lst, $!";

while(<DATA>){
   print "$_";
}
```

Listing 8.41: printing file

Following is the syntax to open emp.lst in writing mode. Here less than > sign indicates that file has to be opend in the writing mode.

```
open(DATA, ">emp.lst") or die "Couldn't open file emp.lst, $!";
```

This example actually truncates (empties) the file before opening it for writing, which may not be the desired effect. If you want to open a file for reading and writing, you can put a plus sign before the > or < characters.For example, to open a file for updating without truncating it.

```
open(DATA, "+<emp.lst"); or die "Couldn't open file emp.lst, $!";
```

To truncate the file first

```
open (DATA, "+>emp.lst" or die "Couldn't open file emp.lst, $!)";
```

You can open a file in the append mode. In this mode writing point will be set to the end of the file.

```
open(DATA,"»emp.lst") || die "Couldn't open file emp.lst, $!";
```

A double » opens the file for appending, placing the file pointer at the end, so that you can immediately start appending information. However, you can’t read from it unless you also place a plus sign in front of it.

```
open(DATA,"+»emp.lst") || die "Couldn't open file emp.lst, $!";
```

8.10.2 Sysopen Function

The **sysopen** function is similar to the main open function, except that it uses the system **open()** function, using the parameters supplied to it as the parameters for the system function.For example, to open a file for updating, emulating the +<filename format from open. `sysopen(DATA, "emp.lst", O_RDWR);`
truncate the file before updating
`sysopen(DATA, "emp.lst", O_RDWR|O_TRUNC );`

Table 8.2: The possible values of MODE

Entities	Definition
O_RDWR	Read and Write
O_RDONLY	Read Only
O_WRONLY	Write Only
O_CREAT	Create the file
O_APPEND	Append the file
O_TRUNC	Truncate the file
O_EXCL	Stops if file already exists
O_NONBLOCK	Non-Blocking usability

You can use O_CREAT to create a new file and O_WRONLY- to open file in write only mode and O_RDONLY - to open file in read only mode.

8.10.3 Close Function

To close a filehandle, and therefore disassociate the filehandle from the corresponding file, you use the close(). This flushes the filehandle's buffers and closes the system's file descriptor.

```
    close FILEHANDLE
close
```

If no FILEHANDLE is specified, then it closes the currently selected filehandle. It returns true only if it could successfully flush the buffers and close the file.

```
close(DATA) || die "Couldn't close file properly";
```

8.10.4 Reading and Writing Files

Once you have an open filehandle, you need to be able to read and write information. There are a number of different ways of reading and writing data into the file.

The <FILEHANDL> Operator

The main method of reading the information from an open filehandle is the <FILEHANDLE> operator. In a scalar context, it returns a single line from the filehandle. For example

```
#!/usr/bin/perl

print "What is your name?\n";
$name = <STDIN>;
print "Hello $name\n";
```

Listing 8.42: Filehandle STDIN

When you use the <FILEHANDLE> operator in a list context, it returns a list of lines from the specified filehandle. For example, to import all the lines from a file into an array

```
#!/usr/bin/perl
print "Enter file name\n";
$filename=<STDIN>;
$filename="<".$filename;
open(DATA,$filename) or die "Can't open data";
@lines = <DATA>;
print @lines;
```

```
close(DATA);
```

Listing 8.43: Filehandle to store file in the array

getc()

The getc function returns a single character from the specified FILEHANDLE, or STDIN if none is specified.

```
getc FILEHANDLE
getc
```

If there was an error, or the filehandle is at end of file, then undef is returned instead.

read()

The read function reads a block of information from the buffered filehandle: This function is used to read binary data from the file.

```
read(FILEHANDLE, SCALAR, LENGTH[, OFFSET]);
```

The length of the data read is defined by LENGTH, and the data is placed at the start of SCALAR if no OFFSET is specified. Otherwise data is placed after OFFSET bytes in SCALAR. The function returns the number of bytes read on success, zero at end of file, or undef if there was an error.

print()

For all the different methods used for reading information from filehandles, the main function for writing information back is the print function.

```
print(FILEHANDLE LIST)
```

The print function prints the evaluated value of LIST to FILEHANDLE, or to the current output filehandle (STDOUT by default).

Copying Files

Here is the example, which opens an existing file emp.lst and read it line by line and generate another copy file emp2.lst.

```
#!/usr/bin/perl

# Open file to read
open(DATA1, "<emp.lst");

# Open new file to write
open(DATA2, ">emp2.lst");

# Copy data from one file to another.
while(<DATA1>)
{
   print DATA2 $_;
}
close( DATA1 );
close( DATA2 );
```

Listing 8.44: copying files

8.10.5 Renaming a file

Here is an example, which shows how we can rename a file file1.txt to file2.txt. Assuming file is available in /usr/test directory.

```
rename ("/usr/test/file1.txt", "/usr/test/file2.txt" );
```

8.11 Regular Expressions and Substitution

8.11.1 tr and s

s: works as it works wit **sed** substitution
tr: Translate

```
while(<>)
{
  @column=split (/\|/);
  $column[1]=~ tr/a-z/A-Z/  ;
  $column[0]=~ s/^/9/;
  $record=join("\|",@column);
  print $record;

}
```

Listing 8.45: translate ans substitute

```
92233|A.K. SHUKLA|g.m.|sales|12/12/52|6000
99876|JAI SHARMA|director|production|12/03/50|7000
```

Listing 8.46: perl translate.pl emp.lst | head -n 2

8.11.2 Advance REGEXP of perl

Table 8.3: Additional regular expression sequence used by perl

Symbols	Significance
\w	Matches a word character (same as [a-zA-Z0-9_])
\W	Doesn't match a word character (same as [â-zA-Z0-9_])
\d	Matches a digit (Same as [0-9])
\D	Doesn't match a digit (same as [0̂-9])
\s	Matches a white space character
\S	Doesn't match a white space character
\b	Matce on word boundary
\B	Doesn't match on word boundary

Following program displace the line numbers of empty lines

```
#in this program we search for empty lines using advance regular expressions of perl

while(<>)
{
  # method 1
  if(/^ *$/)
  {
    print  $. ;
    print "\n";
  }
  #method 2
  if(/^\s*$/)
  {
    print $.;
    print "\n"
  }
}
```

Listing 8.47: line numbers of empty lines

8.12 File Handling

8.13 File Test

```
foreach $file (`ls`)
{
  chop($file);
  print "File $file is readable\n"  if -r  $file;
  print "File $file is executable\n"  if -x $file;
  print "File $file is a non-zero size file\n"  if -s $file;
  print "File $file is exist\n"  if -e  $file;
  print "File $file is Text file\n"  if -T $file;
  print "File $file is Binary file\n"  if -B $file;

}
```

Listing 8.48: File test