Control Flow: nested if

```
if <command1>
then
<command set 1>
elif <command2>
then
<command set 2>
...
else
<command set N>
fi
```

```
var=$(whoami)
if [  $var == one ];
then
   echo "You are admin";
elif [ $var == two ];
then
   echo "you are user";
else
   echo "you are $var";
fi
```

example: nested if

#For each file in /proc/$$/fd/* and ~/li*, loop

```
for file in /proc/$$/fd/* ~/li*
do
    echo -n $file
    if [ -f $file ]
    then
        echo “ is a file”
    elif [ -d $file ]
    then
        echo “ is a directory”
    else
        echo “ is special”
    fi
done
```

```
/proc/5980/fd/0“ is special”
/proc/5980/fd/1“ is special”
/proc/5980/fd/2“ is special”
/proc/5980/fd/255“ is special”
/proc/5980/fd/3“ is special”
/home/rekha/lisp“ is a directory”
/home/rekha/list“ is a file”
/home/rekha/list1.gz“ is a file”
/home/rekha/list.gz“ is a file”
```

Control Flow: multi way selection

- case matches expression with pattern1 first.
- If matched, it executes command1. Otherwise, proceeds to pattern2 and so on.
- Pattern may be a regex (wilcards + EREs).

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

```
case $# in
1) echo "only one arg are passed to $0: $1";;
2) echo "only two args are passed to $0: $1 $2";;
*) echo "more that 2 args passed to $0: $*";;
esac
```

```
$ ./switch.sh foo
only one arg are passed to ./switch.sh: foo
$ ./switch.sh foo bar
only two args are passed to ./switch.sh: foo bar
$ ./switch.sh foo bar baz
more that 2 args passed to ./switch.sh: foo bar baz
```

Control Flow: while

The while command executes a set of commands as long as the condition is true.

```
while CONDITION ; do

    COMMANDS ;

done
```

The condition is to stay in the loop.

```
$ a=1
$ while [ $a -lt 4 ] ;
  do
    echo ${a}${a}_end;
    let a=$a+1;
#use ((a +=1)) or #((a=$a+1)) or let a=$a+1
  done
```

```
11_end
22_end
33_end
```

Control Flow: until

```
until <condition>
do
<commands>
Done
```

<condition> is to come out of the loop.

```
$i=1
$until [ $i -ge 11 ]
   do
      echo $i
      let i+=1
   done
$
```

```
1
2
3
4
5
6
7
8
9
10
```

Control Flow: for

- The for command executes a set of commands for every word in a list of words.
- The commands are executed once for each entry in the words list. Each time the variable specified is equal to the current word.

```
for VARIABLE in LIST OF WORDS ; do

COMMANDS ;

done
```

```
$ for filename in l* ;
    do
        echo ${filename}_brain;
   done
```

```
lab_brain
less_brain
lisp_brain
list_brain
list1.gz_brain
list.gz_brain
```

Control Flow: ranges

Specifying ranges in for loop.
{START..END..INCREMENT}
seq START INCREMENT END

```
$for i in {1..5}
do
  echo $i
done
```

```
1
2
3
4
5
```

```
$for i in {1..10..2}
do
  echo $i
done
```

```
1
3
5
7
9
```

```
$for i in $(seq 1 0.3 2)
do
  echo $i
done
```

```
1.0
1.3
1.6
1.9
```

C-like flavor of for loop

```
$for (( i=1; i<=5; i++))
do
  echo $i
done
```

Control Flow: break

To exit loops prematurely:

```
while <condition>
do
<action 1>
<action 2>
if <some check>
then
break
fi
<action 3>
<action 4>
done
```

```
$ a=1
$ while [ $a -lt 4 ] ;
do
   echo ${a}${a}_end;
   ((a=$a+1));
   if [ $(uname) == Linux ] && [ $a -eq 2 ];
   then
     break;
   fi
done
```

```
11_end
```

Control Flow: continue

Skips to the next loop iteration:

```
for i in <some list>
do
<command 1>
<command 2>
if <some check>
then
continue
fi
<command 3>
done
```

```
$ a=1
#skipping print for a=2
$ while [ $a -lt 4 ] ;
do
   if [ $(uname) == Linux ] && [ $a -eq 2 ];
   then
      ((a=$a+1));
      continue;
   fi
   echo ${a}${a}_end;
   ((a=$a+1));
done

11_end
33_end
```

Control Flow: loop redirection

Input Redirection with Looping

```
$cat >test
foo
bar
baz
```

```
while read var;
do
    echo “${var}”;
done < test
```

```
“foo”
“bar”
“baz”
```

Output Redirection with Looping

```
$for i in {1..2}; do echo "$i"; done > out.dat

$for ((i = 1; i <= 2; ++i )); do echo "$i"; done > out.dat

$for i in 1 2; do echo "$i"; done > out.dat
```

Control Flow: loop redirection

Pipe to Loop

```
$ls li* | while read file
do
   echo hello ${file}
done
```

```
hello list
hello list1.gz
hello list.gz
hello
hello lisp:
hello geiser
```

Loop to Pipe

```
$ for x in li* ;
     do
        [ -r $x ] && echo $x ;
    done | tr 'a-z' 'A-Z'
```

```
LISP
LIST
LIST1.GZ
LIST.GZ
```

Shells - Running a Program

- To run a program type its filename or pathname
  ```
  ls
  ```
- To run two programs sequentially separate by a **‘;’**
  ```
  cd $HOME ; ls
  ```
- To run a program in a sub-shell enclose in **‘(‘ and ‘)’**
  ```
  (cd $HOME ; ls)
  ```
- To run a program in the background append **‘&’**
  ```
  ls / &
  ```

function

- Like other languages, functions can be defined in shell scripts.
- Useful for splitting up scripts into understandable, reusable pieces.
- Functions can be called like independent scripts.

Syntax:

```
function NAME { COMMANDS ; }
```

or the short form:

```
NAME () { COMMANDS ; }
```

```
$ function hi() { echo "hello there"; }
$ hi
hello there
```

Regular Expression

Regular expressions are a form of pattern matching syntax which many commands use. (e.g. grep, sed)

A Regular Expression contains one or more of the following:

A character set: These are the **characters** retaining their literal meaning. The simplest type of Regular Expression consists only of a character set, with no metacharacters.

An anchor: These designate (anchor) the **position** in the line of text that the RE is to match. For example, ^, and $ are anchors.

Modifiers: These expand or narrow (modify) the **range** of text the RE is to match. Modifiers include the asterisk, brackets, and the backslash.

Regular Expression

- A specific search pattern entered to find a particular target string.
- They are very flexible and not quickly learnt.
- Some basic forms are easy to learn and very useful.
- Special characters used in regular expressions include:

.	**matches any one character**
*****	**matches zero or more of the last character**
.*	**matches any string**
[ ]	**matches any character in the range**
^	**represents the start of the line**
$	**represents the end of the line**
<	**matches start of a word.**
• >	**matches end of a word.**
[^]	**matches any character not in the range**

Regular Expression

- Interpretted by the command, and not by the shell.
- Not the same as shell wildmasks, although some are similar.
- Regex metacharacters overshadow the shell’s.

RE **. => ?** in shell

RE *** => 0+** occurrence of previous char

RE **.* => *** in shell means none or any char

Regular Expression

```
grep '^[aeiou].*' /usr/share/dict/words #begin with vowels
ls |egrep '[^a-c]..[a-c]' #4 letter word begins with non a,b,c and ends with a/b/c
ls -l | grep '^-rw-' #begins with -rw-
^$ #begin end: blank line
grep '^bash' /usr/share/dict/words #begins with bash
grep ‘shell$’ /usr/share/dict/words #ends with shell
grep '\<computer' /usr/share/dict/words
        #escaping < to see the beginning of word
grep 'computer\>' /usr/share/dict/words
        #escaping > to see the end of the word
sudo ls /proc/1/fd | grep '^[[:digit:]]$'
#all files which have single digit as filename
```

Regular Expression:metachars

More readable Named Character Classes exist in dealing with more complex expressions.

- [:alnum:] - alphanumeric characters; same as [a-zA-Z0-9]
- [:alpha:] - alphabetic characters; same as [a-zA-Z]
- [:digit:] - digits; same as [0-9]
- [:upper:] - upper case characters; same as [A-Z]
- [:lower:] - lower case characters; same as [a-z]
- [:space:] - any white space character, including tabs.
- [:punct:] - Punctuation characters.

```
ls -l | grep [[:digit:]] #display filenames containing digit
ls  | grep '^[a[:digit:]b]' #all files which start with digit or ‘a’ or ‘b’
```

Regular Expression: grep

```
grep "^mo.*ing$" /usr/share/dict/words
#begins with mo followed by any number of chars and ending with ing
grep '^e.*l\+y' /usr/share/dict/words
#begins with e, contains ly,lly,llly,...
 egrep '^e.*l+y' /usr/share/dict/words
# same as above, no escape for + in extended regular expression format
grep '^[[:upper:]].*w$' /usr/share/dict/words
#begins with upper case char and ends with w
grep '^[[:upper:]a].*w$' /usr/share/dict/words
#begins with either upper case char or ‘a’ and ends with w
```

Regular Expression

- Most of the metachars must be escaped (in BRE)!
- Asterisk/Kleene star (*) - matches 0+ occurence(s) of an expression.
- Optional (\?) - matches 0 or 1 occurence of an expression
- Alternation (\|) - matches either of the expressions it sits between.
- Plus (\+) - matches 1+ occurrence(s) of an expression

d* M[sr]\|Miss

Saviou\?r ho\+ray

- To avoid escaping, use egrep or grep -e to use ERE instead BRE.

sed

- The sed command performs string substitutions. It is usually used to add, remove or change parts of a string. This is often invaluable for modifying variables.

[address-range]/p **print**

[address-range]/d **delete**

s/pattern1/pattern2/ **substitute** pattern2 for first instance of pattern1 in a line

[address-range]/s/pattern1/pattern2/ **substitute** pattern2 for first instance of pattern1 in a line, over address-range

[address-range]/y/pattern1/pattern2/ **transform any character** in pattern1 with the corresponding character in pattern2, over address-range (equivalent of tr)

[address] i pattern Filename **Insert pattern** at address indicated in file Filename.

g Operate on every pattern match within each matched line of input

sed

- The sed command performs string substitutions. It is usually used to add, remove or change parts of a string. This is often invaluable for modifying variables.

[address-range]/p **print**

[address-range]/d **delete**

s/pattern1/pattern2/ **substitute** pattern2 for first instance of pattern1 in a line

[address-range]/s/pattern1/pattern2/ **substitute** pattern2 for first instance of pattern1 in a line, over address-range

[address-range]/y/pattern1/pattern2/ **transform any character** in pattern1 with the corresponding character in pattern2, over address-range (equivalent of tr)

[address] i pattern Filename **Insert pattern** at address indicated in file Filename.

g Operate on every pattern match within each matched line of input

Sed : Substitution

- Syntax for changing STRING1 to STRING2 is:

  sed s/STRING1/STRING2/g
- Any character can be used instead of / e.g.

  sed s@STRING1@STRING2@g
- The characters **. * [ ] /** have special meaning in the first string unless preceded by a backslash
- A quick way to double-space a text file is **sed G filename.**

Sed : examples (substitution/delimiter)

```
$v=`ls l* | sed s/li/Pl/g`
$ echo $v
lab less Plst Plst1 Plst1.gz Plst.gz Plsp:
geiser
```

```
$ v=`ls l* | sed s@li@Pl@g`
$ echo $v
lab less Plst Plst1 Plst1.gz Plst.gz Plsp:
geiser
```

Examples(use of regex in pattern)

```
• $ echo "Hello world" | sed 's/w.*/X/g'
  Hello X
• $ echo "Hello world" | sed 's/w*/X/g'
  XHXeXlXlXoX XXoXrXlXdX
• $ echo "Hello world" | sed 's/^.* /X/g'
  Xworld
• $ echo "Hello world" | sed 's/.$/X/g'
  Hello worlX
• $ echo "Hello world" | sed 's/[wo]/X/g'
  HellX XXrld
• $ echo "Hello world" | sed 's/[^wo]/X/g'
  XXXXoXwoXXX
```

Examples (insert and transform)

- ls i*|sed '**y/abcd/ABCD/'**

  #replace each char in abcd with corresponding char in ABCD

- ls i*|sed **'i\this is an inserted line\'**

  #inserts line

- ls i*|sed **'3i\Linux is great.\'**

  #Inserts line 'Linux is great.' at line 3

- echo "Working on it." | sed -e **'1i How far are you along?'**

  #Prints "How far are you along?" as first line, "Working on it" as second.

Examples (delete and print)

-
  ```
  ls i*|sed  '2d'
  ```
  #Delete 2th line of input.

-
  ```
  sed '/^$/d' test
  ```
  #Delete all blank lines from test

-
  ```
  sed -e '1,15{/^$/d}' test
  ```
  Delete blank lines within range of 1 to 15.

-
  ```
  ls *|sed -n '/.*arg.*/p'
  ```
  # use -n to print only those lines matching the pattern

checkpoint

- `s/Windows/Linux/`
  Substitute "Linux" for first instance of "Windows" found in each input line.
- `s/BSOD/stability/g`
  Substitute "stability" for every instance of "BSOD" found in each input line.
- `s/ *$//`
  Delete all spaces at the end of every line.
- `s/00*/0/g`
  Compress all consecutive sequences of zeroes into a single zero.
- `s/GUI//g`
  Delete all instances of "GUI", leaving the remainder of each line intact.
- `/GUI/d`
  Delete all lines containing "GUI".
- `/Jones/p`
  Print only lines containing "Jones" (with -n option).

awk

- The awk command is a very general pattern matching facility.
- One simple but useful capability is to pick out columns of text.
- This is particularly handy for manipulating tabular information.
- Named after its authors: Alfred Aho, Peter Weinberger, and Brian Kernighan.
- A powerful programming language for text manipulation + report writing (precursor to perl).
- C-like syntax (functions, arrays, if, for & while constructs, etc).
- Combines features from many filters (e.g. grep, sed).
- Regex-aware (ERE)
- Flavors: new awk (nawk), GNU awk (gawk), ...

awk

awk [options] ‘pattern {action}’ file(s)

Searches for pattern and applies action on it.

- Default action is to print current record on STDOUT.
- Default pattern is to match all lines.
- If file(s) not specified, input taken from??

Common options:

- -f read program/pattern from a file
- -F sets field separator (FS) value (default is “ ”)

awk

Syntax for selecting column N is:
  awk '{print $N}'

Note that the exact syntax (quotes and braces) must be used.

```
$ls -l * | awk '{print $9}' #print 9th column
$ls -l * | awk '{print $1,$9}' #print 1st and 9th column
$ls -l *| awk '{print $1,"\t",$9}' #add tab between columns
$ls -l *| awk '/so+/{print $1,"\t",$9}' #only rows which satisfy RE
$ls -l *| awk '{print length($0), "\t",$1,"\t",$9}' #$0 gives line to be edited
```

```
#begin block to be run once in the beginning of text processing
#body block is for each input line
#end block to be run once in the end of text processing:last line
```

```
awk 'BEGIN {for (i=1;i<ARGC;i++)str=sprintf("%s %s",str,ARGV[i]);}
             {print $1 "\t" str}
      END {print "This is the end!"}' foo bar baz
```

Explore Further

Google is your best friend.

Linux shell scripting tutorial http://www.freeos.com/guides/lsst/

Advanced Bash Scripting http://tldp.org/LDP/abs/html/

Unix shell scripting http://www.tutorialspoint.com/unix/