Course code: 4IS05

UNIT-4

Essential Shell Programming

Prepared by : Sushma S.A
Department :Information Science

Date : 06-04-2020

Essential Shell Programming

- A shell program runs in interpretive mode.
- Each statement is loaded into memory when it is to be executed.
- shell scripts run slower than those written in high-level languages.

Shell Scripts:

- the shell is also a programming language that executes shell scripts in the ***interpretive mode*** – one line at a time.
- the ***interpreter line*** signifies the sub-shell that runs the script.
- 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.**

- normally **.sh** extension is used for shell scripts.
- shell scripts are executed in a separate **child shell process**, and this sub-shell need not be of the same type as the login shell.
- by default, the child and parent shells belong to the same type.

To create a shell script:

- Use **vi** editor to create the shell script , **script.sh**. The following script runs 3 echo commands and shows the use of variable evaluation and command substitution

```
#!/bin/sh
 # script.sh: Sample shell script
echo “Today’s date : `date`”
 echo  “This month’s calendar :”
 cal  `date   “+%m  20%y”`
echo “My shell : $SHELL”
```

- **#** is the **comment character** that can be placed anywhere in a line; the shell ignores all characters placed on its right. But this is not true with the first line.
- The very first line is a **interpreter line** , begins with **#!** and this is followed by the path name of the shell to be used for running the script.

To run the script, make it executable first and then invoke the script name as shown :

```
$ chmod  +x  script.sh
 $ script.sh
  Today's  date: Mon Apr 6 10:02:42   IST 2020
  This month's calendar:
       April 2020
 Su Mo Tu We Th Fr Sa
              1 2 3 4
……………………..
    My shell : /bin/sh
```

read : Making Scripts Interactive

- The **read** statement is used with one or more variables to provide input to a script from the keyboard.
- For example,

```
read name
```

The script pauses at that point to take input from the keyboard. Whatever is entered is stored in the variable ***name***.

```
#!/bin/sh
# emp.sh: Interactive version –uses to read to take two inputs
 #
 echo “Enter the pattern to be searched: \c”
read pname
echo “Enter the file to be used :  \c”
 read flname
 echo “Searching for $pname from file $flname”
 grep “$pname” $flname
 echo “Selected records shown above”
```

- Run the script and specify input . The script first asks for a pattern to be entered, which is then assigned to the variable *pname*. Next, the script asks for the filename , which is assigned to the variable *flname*. Then **grep** runs with these two variables as arguments.

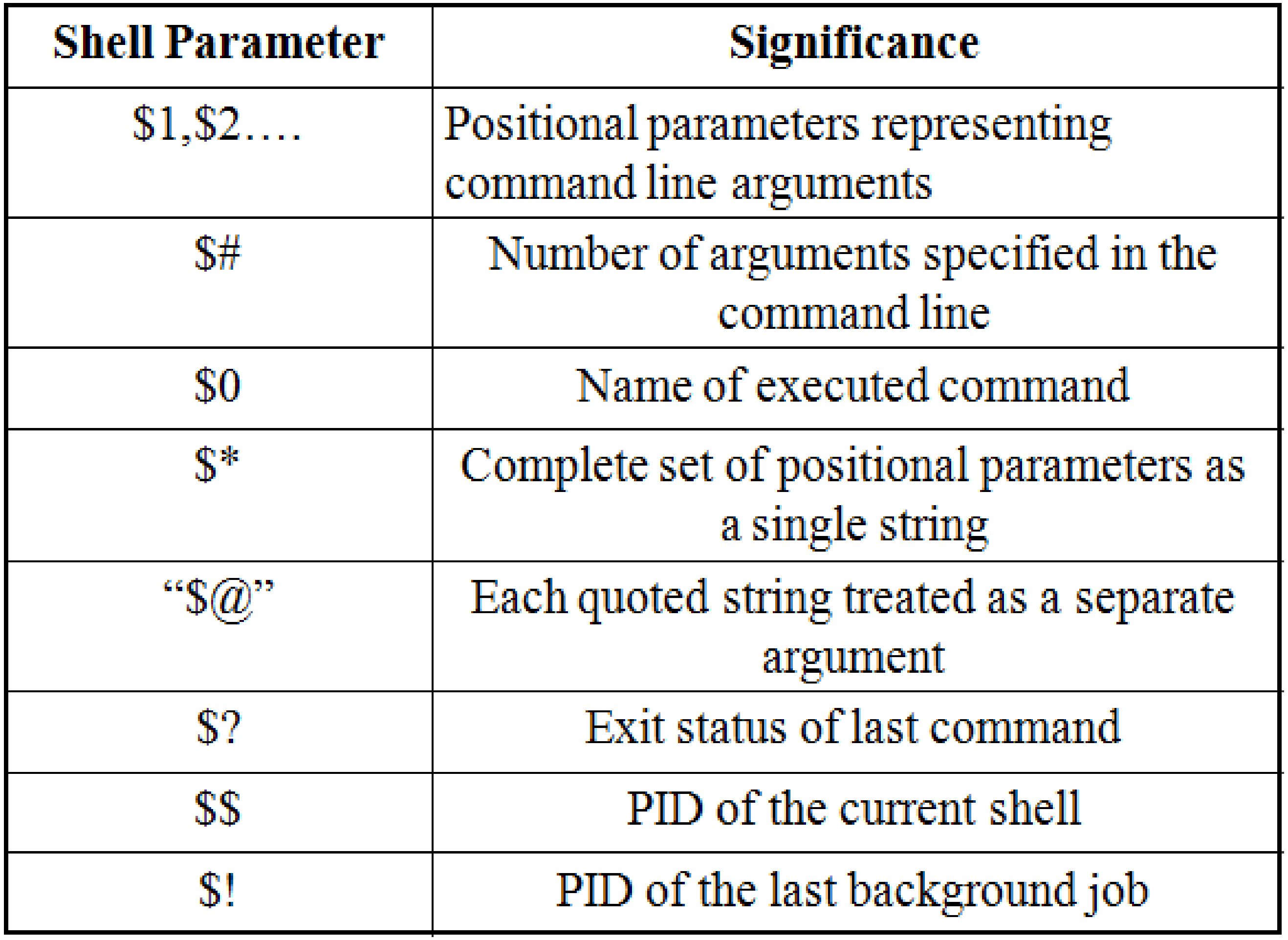
□ A single **read** statement can be used with multiple arguments.

read pname flname

Using Command Line Arguments:

- shell scripts accept arguments from the command line. They run non interactively and can be run with redirection and pipelines.
- command line arguments passed to a script are read into **positional parameters** (like $1, $2 etc.). The first argument is read by the shell into the parameter $1, the second argument into the $2, and so on.
- **$*** : it stores the complete set of positional parameters as a single string.
- **$#** : it stores the number of arguments.
- **$0** : contains the name of the script itself.

Special parameters used by the shell



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

```
#!/bin/sh
# emp2.sh: Non-interactive version-uses command line
    arguments
#
 echo “Program: $0              # $0 contains program name
 The number of arguments specified is $#
  The arguments are   $*”       # all arguments stored in $*
  grep “$1”   $2
  echo “\n job over”
```

exit and EXIT STATUS OF COMMAND:

- The **exit** statement terminates a script.
- 2 common exit status values are:

exit 0 *used when everything went fine*

exit 1 *used when something went wrong*

- through the exit command or function, every command returns an exit status to the caller. A command is said to return a *true exit status* if it executes successfully, and *false* if it fails.
- **$ cat foo**

cat: can’t open foo

- Returns a nonzero exit status (i.e.**1**) because it couldn't open the file.

The Parameter $?:

- the parameter **$?** stores the exit status of the last command.
- It has the value 0 if the command succeeds and a nonzero value if it fails.
- this parameter is set by **exit’**s argument. If no exit status is specified, then **$?** is set to zero (true).

```
$  grep director emp.lst  >/dev/null ;  echo $?
0                                success
$  grep manager emp.lst  >/dev/null ;  echo $?
1                                Failure-in finding pattern
 $  grep manager emp3.lst  >/dev/null ;  echo $?
 grep: can’t open emp3.lst               Failure-in opening a file
 2
```

The Logical operators && and || - Conditional Execution:

- the && and || are used as simple conditionals.
- syntax is : **cmd1 && cmd2**

  cmd1 || cmd2

The **&&** delimits two commands ; the command ***cmd2*** is executed only when ***cmd1*** succeeds.

```
$ grep ‘director’ emp.lst  && echo “pattern found in file”
1006 |xyz  gupta  | director | sales |03/06/56 |6700
6521 |lalit  choudhary | director| marketing | 04/07/45 |8600
```

With the **||** operator, the second command is executed only when the first fails.

```
$ grep ‘manager’ emp.lst  ||  echo “Pattern not found”
Pattern not found
```

The if conditional:

the if statement makes two-way decisions depending on the fulfillment of a certain condition. In the shell, the if statement uses the following forms:

Form1:

```
if command is successful
 then
    execute commands
else
     execute commands
fi
```

Form2:

```
if command is successful
then
    execute commands
fi
```

Form 3:

```
if command is successful
then
    execute commands
elif command is successful
then…
else…..
fi
```

- The if statement evaluates the success or failure of the command that is specified in its “command line”.
- Consider the script below:

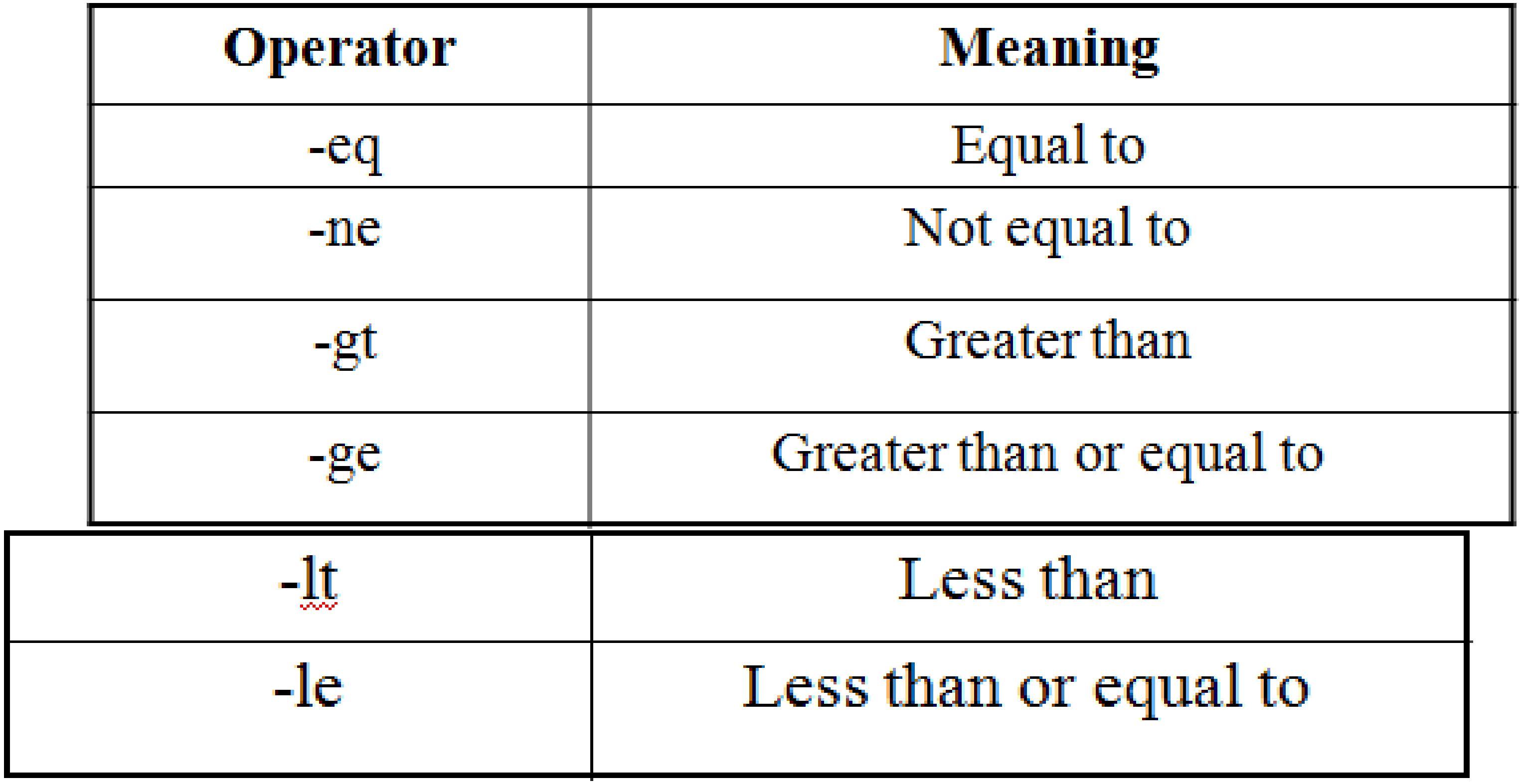
```
#! /bin/sh
# emp3.sh :using if and else
#
if grep “^$1” /etc/passwd   2>/dev/null   #search username at beginning of line
then
        echo “Pattern found –job over”
else
        echo “ Pattern not found”
fi
```

```
$  emp3.sh   ftp
 ftp:  *:  325:15:FTP  user:/users1/home/ftp:/bin/true
 Pattern found-job over
  $ emp3.sh  mail
  Pattern not found
```

Using *test* and [] to evaluate expressions:

- Test can be used with operators to compare numbers and strings , as well as to check the various file attributes.
- test uses certain operators to evaluate the condition on its right and either true or false exit status, which is then used for making decisions.
- **test** works in 3 ways:
  - compares two numbers
  - compares two strings or a single one for a null value
  - checks a file’s attribute
- test doesn’t display any output but simply sets the parameter **$?.**

Numeric Comparison:



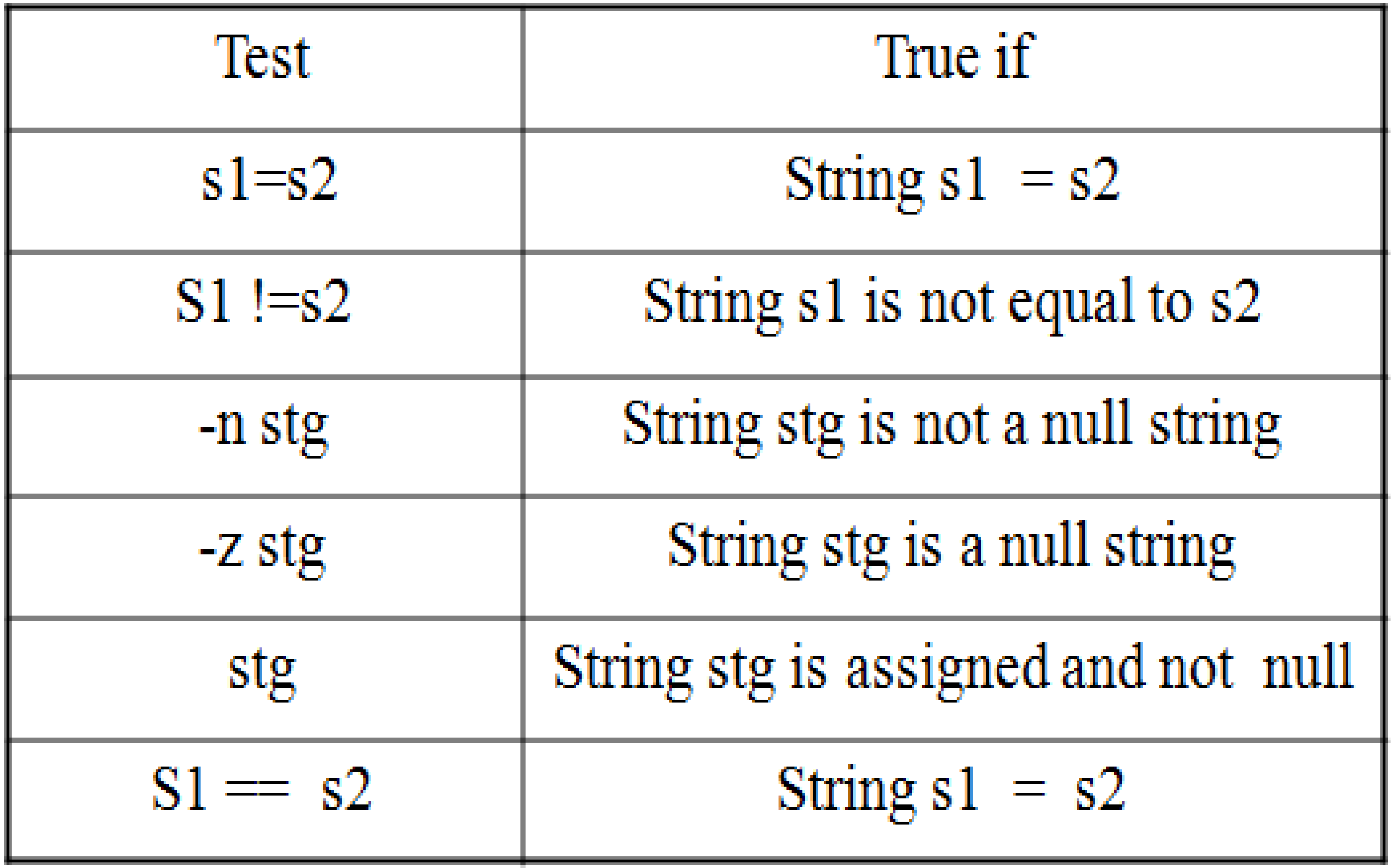
Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to

- numeric comparison in shell is confined to integer values only; decimal values are simply truncated.

```
$ x=5; y=7; z=7.2
$ test $x –eq  $y  ;  echo  $?
1
$ test $x  –lt $y ; echo $?
0
$ test $z  -gt   $y ; echo $?
 1
   $ test $z  -eq  $y ; echo $?
    0
```

```
#!/bin/sh
# emp3a.sh: Using test, $0 and $# in an if-elif-if construct
#
 if test $#  -eq  0  ; then
       echo “Usage : $0 pattern file “ > /dev/tty
 elif test $#  -eq 2 ;   then
       grep “$1” $2  ||  echo  “$1 not found in $2” >/dev/tty
 else
        echo “ You didn’t enter two arguments” >/dev/tty
 fi
```

String Comparison:



Test	True if
s1=s2	String s1 = s2
S1 !=s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not null
S1 == s2	String s1 = s2

```
#! /bin/sh
# emp4.sh: checks user input for null values
#
 if [$#  -eq  0]  ; then
     echo “enter the string to be searched:  \c”
     read pname
       if [-z  “$pname”]  ;  then
        echo “ You have not entered the string”   ;   exit   1
       fi
     echo “ enter the filename to be used :  \c”
      read flname
     if [ ! –n “$flname”]    ;    then
echo “You have not entered the filename”  ;  exit 2
  fi
    emp3a.sh  “$pname” “$flname”
  else
     emp3a.sh  $*
 fi
```

now run the script:

```
 $ emp4.sh
 Enter the string to be searched : [Enter]
 You have not entered the string
 $ emp4.sh
 Enter the string to be searched : root
Enter the filename to be used : /etc/passwd
 root:x:0:1:Super-user:/:/usr/bin/bash
                        #from emp3a.sh
```

Now run the script with arguments. **emp4.sh** bypasses all of the above activities and calls emp3a.sh to perform all validation checks:

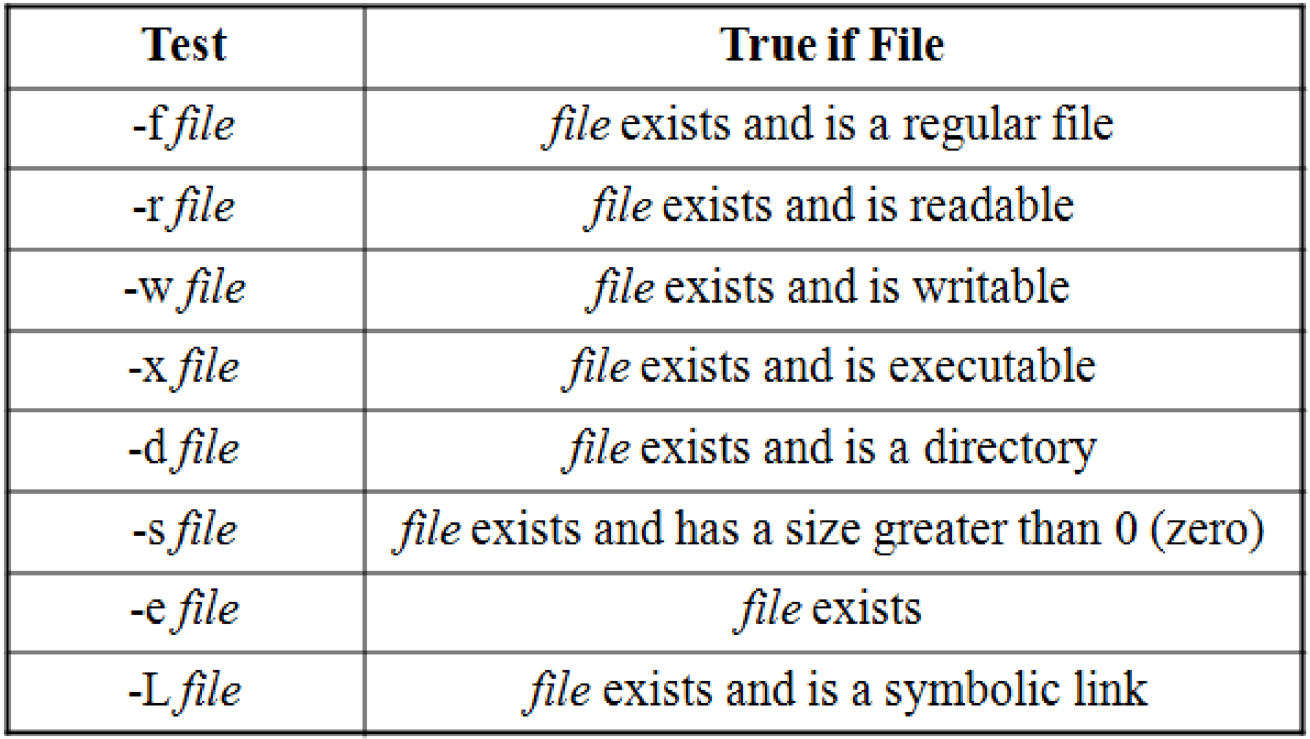
```
$ emp4.sh jai
You didn’t enter two arguments
$ emp4.sh jai emp.lst
9876 | jai      sharma |     director | production | 12/03/50 |7000
$ emp4.sh “jai sharma” emp.lst
You didn’t enter two arguments
```

File Tests:

- **test** can be used to test the various file attributes like its type (file, directory or symbolic link) or its permissions (read, write, execute etc.).

```
$  ls   -l  emp.lst
-rw-rw-rw-    1   kumar    group    870   Jun 8  15:45    emp.lst
$ [  -f  emp.lst]   ;  echo  $?
0
$ [ -x  emp.lst]   ;  echo $?
1
$ [  !  -w emp.lst]  ||  echo “false that file is not writable”
false that file is not writable
```

File-related tests with test



Test	True if File
-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 a directory
-s *file*	*file* exists and has a size greater than 0 (zero)
-e *file*	*file* exists
-L *file*	*file* exists and is a symbolic link

- Write a script that accepts a filename as argument and then performs a number of tests on it:

```
#! /bin /sh
# filetest.sh: tests file attributes
 if [ !  -e  $1] ;  then
       echo “ File doesn’t exist”
 elif [ ! –r  $1]  ;  then
      echo “ File is not readable”
 elif [ !    -w $1] ; then
      echo “File is not writable”
 else
      echo “File is both readable and writable”
Fi
```

- Now run this script:
- **$ filetest.sh emp3.lst**
- File does not exist
- $ filetest.sh emp.lst
- File is both readable and writable

The case CONDITIONAL:

- The statement matches an expression for more than one alternative , and uses a compact construct to permit multiway branching.
- The general syntax of the case statement is as follows:

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

- case first matches *expression* with *pattern1.* If the match succeeds, then it executes *commnads1* , which may be one or more commands.
- If the match fails, then *pattern2* is matched , and so forth.
- Each command list is terminated with a pair of semicolons, and the entire construct is enclosed with **esac** (reverse of case).

```
#!/bin/sh
# menu.sh: uses case to offer 5-item menu
#
 echo  “ MENU\n
  1. list of files \n  2. Processes of user \n  3.Today’s date
  4. Users of system \n 5. Quit to UNIX\n Enter your option:  \c”
   read choice
   case “$choice” in
   1) ls –l   ;;
   2) ps –f  ;;
   3) date  ;;
4)     Who  ;;
5)      exit   ;;
  *) echo “Invalid option”
 Esac
```

The last option (*) matches any option not matched by the previous options.

Running the script:

```
$ menu.sh
       MENU
1.   List of files
2.    Processes of user
3.    Today’s date
  4.   Users of system
  5.   Quit to UNIX
Enter your option:  3
Sat Nov 8  09:30:45   IST 2008
```

Matching Multiple Patterns:

- The case statement can also specify the same action for more than one pattern. For example, to test a user response for both **y** and **Y.**
- The expression **y|Y** is used to match y in both uppercase and lowercase.

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

expr: Computation and String Handling

The expr command performs following two functions:

- Performs arithmetic operations on integers.
- Manipulates strings.

Computation:

- **expr** can perform the four basic arithmetic operations as well as the modulus function

Computation:

- **expr** can perform the four basic arithmetic operations as well as the modulus function

```
$ x=3  y=5
$ expr 3 + 5
8
$ expr $x - $y
-2
$ expr 3  \* 5       //asterisk has to be escaped
15
$ expr $y / $x       // decimal portion truncated
1
$ expr 13 % 5
3
```

- the operands +, -, * etc. must be enclosed on either side by whitespace.
- **expr** is often used with command substitution to assign a variable. We can set the variable z to the sum of two numbers:

```
$ x=6 y=2; z= `expr $x + $y`
$ echo $z
8

$ x=5
$ x=`expr $x + 1`
$ echo $x
6
```

String Handling:

- For manipulating strings, **expr** uses two expressions separated by a colon. The string to be worked upon is placed on the left of the **;**, and a regular expression is placed on its right.
- Depending on the composition of the expression, expr can perform three important string functions:
- **Determine the length of the string**
- **Extract a substring**
- **Locate the position of a character in a string**

Length of a string:

- the regular expression .* signifies to expr that it has to print the number of characters matching the pattern , i.e. , the length of the entire string

```
$ expr  “abcdefghijkl”   :  ‘.*’
12
```

- Here**, expr** has counted the number of occurrences of any character (.*) .

Extracting a substring:

- expr can extract a string enclosed by the escaped characters \ (and \).
- to extract a 2-digit year from a 4-digit string:

```
$ stg=2003
 $ expr “$stg”  :  ‘..\(..\)’          # extracts last two characters
 03
```

Locating Position of a character:

- expr can also return the location of the first occurrence of a character inside a string.
- to locate the position of the character **d** in the string value of **$stg** ,
- we have to count the number of characters which are not **d ([^d]*),** followed by a **d**:

```
$ stg=abcdefgh ; expr “$stg”  :’[^d]*d’
4
```

$0: Calling a Script by Different Names

- There are a number of UNIX commands that can be used to call a file by different names and doing different things depending on the name by which it is called. $0 can also be to call a script by different names.

Example:

```
#! /bin/sh
#
lastfile=`ls –t *.c |head -1`
command=$0
exe=`expr $lastfile: ‘\(.*\).c’`
case $command in
*runc) $exe ;;
*vic) vi $lastfile;;
*comc) cc –o $exe $lastfile &&
        Echo “$lastfile compiled successfully”;;
esac
```

After this create the following three links:

```
ln comc.sh comc
 ln comc.sh runc
ln comc.sh vic
```

Output:

```
$ comc
hello.c  compiled successfully
```

While: Looping

To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

Synatx:

```
while condition is true
do
    Commands
done
```

The commands enclosed by do and done are executed repadetedly as long as condition is true.

Example:

```
#! /bin/usr ans=y
while [“$ans”=”y”]
do
        echo “Enter the code and description : \c” > /dev/tty
        read code description
        echo “$code $description” >>newlist
         echo “Enter any more [Y/N]”
        read any
        case $any in
                Y* | y* ) answer =y;;
                N* | n*) answer = n;;
                *) answer=y;;
        esac
done
```

Input:

```
Enter the code and description : 03 analgestics
Enter any more [Y/N] :y
Enter the code and description : 04 antibiotics
Enter any more [Y/N] : [Enter]
Enter the code and description : 05 OTC drugs
Enter any more [Y/N] : n
```

Output:

```
$ cat newlist
03 | analgestics
04 | antibiotics
05 | OTC drugs
```

Other Examples: An infinite/semi-infinite loop

```
(1) (2)
while true ; do while [ ! -r $1 ] ; do
[ -r $1 ] && break sleep $2 sleep $2 done
done
```

for: Looping with a List

The syntax is:

```
 for variable in list
 do
      commands
 done
```

The loop body also uses the keywords **do** and **done** .

```
$ for file in chap20 chap21 chap22 chap23  ;  do
>       cp $file ${file}. bak
>       echo $file is copied to $file.bak
▯     done
 chap20  copied to chap20.bak
 chap21  copied to chap21.bak
 chap22  copied to chap22.bak
 chap23  copied to chap23.bak
```

Possible sources of the list:

List from variables:

- We can use a series of variables in the command line. They are evaluated by the shell before executing the loop:

$ for var in $PATH $HOME $MAIL ; do echo "$var" ; done

```
/ bin : /usr/bin:/home/local/bin:/usr/bin/X11: .: / oracle/bin
/home/henry
/var/mail/henry
```

List from Command Substitution:

- we can also use command substitution to create the list.
- the following **for** command line picks up its list from the file *clist*:

for file in `cat clist`

- this method is most suitable when the list is large and we need not specify its contents individually. Also, we can change the list without having to change the script.

List from Wild-cards:

- when list consists of wild cards (such as *** ,?),** the shell interprets them as *filenames*.
- the **for loop** shown below works on every HTML file in the current directory :

```
for file in *.htm *.html ; do

        sed  ‘s/strong/STRONG/g                # sed  performs
     substitution

      s/img src/IMG SRC/g’ $file >>$$

      mv $$ $file

     gzip $file          #file compression takes place

•    done
```

List from Positional Parameters:

- **for** is also used to process positional parameters that are assigned from command line arguments.

```
# !/bin/sh
# emp6.sh: using a for loop with positional parameters
#
 for pattern in “$@”   ;  do
  grep “$pattren” emp.lst || echo “Pattren $pattren not found”
done
```

- Now, execute this script by passing 4 arguments, one of which is a multiword string:

```
$ emp6.sh  2345  1265  “jai  sharma”  4379
 2345 |j.b  saxena      |g.m.        |marketing  |12/03/45 | 8000
 1265 |s.n.  Dasgupta  |manager |sales          |12/09/63  | 4500
 9976 | jai  sharma     |director    | production |12/03/50 |7000
Pattern 4379 not found
```

basename: Changing Filename Extensions

- we can use **basename** command inside a **for** loop to change the extensions of filenames.
- basename extracts the “base” filename from an absolute pathname:

```
$ basename /home/henry/project3/de2bin.pl
 dec2bin.pl
```

When **basename** is used with two arguments, it removes the second argument from the first argument:

```
$ basename ux2nd.txt  txt
ux2nd .
```

to rename file extension from **txt** to **doc**:

```
for file in *.txt  ;  do
 leftname =`basename $file txt`    # stores left part of filename
 mv $file ${leftname}doc
done
```

For example, if **for** statement picks up **seconds.txt** as the first file, leftname stores **seconds. .** **mv** simply adds a **doc** to the extracted string **(seconds.)** and the file becomes **seconds.doc**

set and shift: Manipulating the Positional Parameters

- **set** places values into positional parameters $1,$2, and so on.
- this is useful for picking up individual fields from the output of a program.

```
  $ set 9876 2345 6213
 $_
```

- this assigns the value 9876 to the positional parameter $1 , 2345 to $2 and 6213 to $3.
- it also sets other parameters **$#** and **$***.

```
$ echo “\$1 is $1, \$2 is $2, \$3 is $3”
$1 is 9876, $2 is 2345, $3 is 6213
$ echo “The $# arguments are $*”
```

- The 3 arguments are 9876 2345 6213

- We can use command substitution to extract individual fields from the **date** output:

```
$ set `date`
$ echo $*
Wed Nov 12 10:30:55   IST 2008
$ echo “The date today is $2 $3,  $6”
The date today is Nov 12 , 2008
```

shift: Shifting Arguments Left

- **shift** transfers the contents of a positional parameter to its immediate lower numbered one. This is done as many times as the statement is called. When called once **$2** becomes **$1** , **$3** becomes **$2**, and so on

```
$ echo "$*"
Wed Nov 12 10:30:55   IST 2008
$ echo $1 $2 $3
Wed Nov 12
$ shift                      #shifts 1 place
$ echo $1 $2 $3
Nov 12 10:30:55
$ shift 2           #shifts 2 places
$ echo $1 $2 $3
10:30:55
```

set --: Helps Command substitution

set - - is recommended for use when using command substitution.

set - - `ls –l unit01`

set - - `grep PPP /etc/passwd`

while :LOOPING

- **loops** let us perform a set of instructions repeatedly. The shell features 3 types of loops: **while, until** and **for.**
- a **while loop** is used for repeatedly executing a group of commands.
- the while loop repeatedly performs a set of instructions until the control command returns a true exit status.
- the general syntax is as follows:

```
while condition is true
   do
      commands
  done
```

- The *commands* enclosed by **do** and **done** are executed repeatedly as long as *condition* remains true.
- Consider the following script:

```
#!/bin/sh
# emp5.sh: Shows use of while loop
#
 answer=y
 while [“ $answer”=“y”]
  do
   echo “ Enter the code and description :\c” >/dev/tty
   read code description
   echo “$code | $description” >> 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
```

Now, run this script:

```
 $ emp5.sh
 Enter the code and description : 03 analgesics
 Enter any more (y/n)? Y
 Enter the code and description: 04 antibiotics
  Enter any more (y/n)? [Enter]
```

```
Enter the code and description: 05 OTC drugs
Enter any more (y/n)? n
```

Now see the contents of file ***newlist***:

```
$ cat newlist
03 | analgesics
04 | antibiotics
05 | OTC drugs
```

Using while to Wait for a File:

Consider the script below:

```
#!/bin/sh
# monitfile.sh: Waits for a file to be created
#
 while [! –r invoice.lst]      # while the file invoce.lst can’t be read
  do
    sleep 60                   # sleep for 60 seconds
  done
 alloc.pl                 # execute this program after exiting loop
```

- This script periodically monitors the disk for the existence of the file, and then executes the program once the file has been located. It make use of the external **sleep** command that makes the script pause for the duration (in seconds) as specified as its argument.
- The loop executes repeatedly as long as the file ***invoce.lst*** can't be read. If the file becomes readable , the loop is terminated and the program alloc.pl is executed.
- **sleep** command is quite useful in introducing some delay in shell scripts.
- this script can be run in background like this :

monitfile.sh &

Setting up an infinite loop:

- The infinite loop is best implemented by using **true** as a dummy control command with **while**. The **true** simply returns a true exit status. Another command **false** returns a false value.

```
while true ; do
    df  –t            # df reports free space on disk
    sleep 300
done &            # & after done runs loop in background
```

- The above script checks the free space available on your disk every 5 minutes. The script is run in the background without interrupting our other work, but every 5 minutes we can see a screen filled with **df** output.

trap: Interrupting a Program

- the shell script terminates whenever an interrupt key is pressed. But this leaves a lot of temporary files on disk.
- the **trap** statement lets us do the things we want in case the script receives a signal.
- the trap statement is normally placed at the beginning of a shell script and uses two lists:
- **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 more signals – the one which is used with **kill** command

- trap ‘rm $$* ; echo “Program interrupted” ; exit’ HUP INT TERM
- **trap** is a signal handler. Here, it first removes all files expanded from $$* , echoes a message and finally terminates the script when the signals SIGHUP (1), SIGINT (2) or SIGTERM (15) are sent to the shell process running the script. When the interrupt key is pressed , it sends the signal number 2.
- We can also make a script to ignore the signal and continue processing. In this case, the script simply ignores such signals by using a null command list.
- A script containing the following **trap** statement will not be affected by three signals:

trap ‘ ‘ 1 2 15

The HERE Document (<<):

- The shell uses the **<<** symbols to read data from the same file containing the script. This is referred to as **here document** , signifying that the data is here rather than in a separate file. Any command using a standard input can also take input from a here document.
- This feature is useful when used with commands that don’t accept a filename as argument . If the message is short, both the command and message can be given in the same script:

Example:

```
    mailx  sharma   <<  MARK
Your program for printing the invoices has been executed
 on `date` . Check the print queue
The updated file is known as $flname
 MARK
```

- The here document symbol (**<<**) is followed by 3 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.
- The user *sharma* at the other end will see the 3 lines of message text with the date inserted by command substitution and evaluated filename.
- When this sequence is placed inside a script, execution is faster because **mailx** doesn’t have to read an external file