read (read from user): Making script interactive

The read statement is the shell's internal tool for taking input from the user. i.e, making scripts interactive. It is used with one or more variable.

```
$ read name

manish

$
```

It will take input from the user and store in variable name.

```
$ echo $name

manish

$
```

Script 1: write a shell script that take input pattern name and file name , and search the pattern in file and display records.

Script Name: script1.sh

```
#!/bin/sh

echo "Pattern to be search : \c"

read pname

echo  "Enter Filename \c"

read fname

grep "$pname"  $fname
```

To run the script sh command is used.

```
$ sh script1.sh

Pattern to be search : bcom
```

```
1114 | Samir P. Chowdhari | 12/02/1982 | Surat     | bcom | 8461626364
1114 | Samir P. Chowdhari | 12/02/1982 | Surat     | bcom | 8461626364
```

Using command line argument

In UNIX, shell script accept arguments from the command line. When arguments are specified with a shell script, they are assigned to certain special variables called positional parameters.

The first arguments read by shell in to parameter $1, second argument in $2 and so on.

$* - It stores the complete set of positional parameter as a single string

$# - It is set to the number of argument specified

$0 - hold the command name itself.

Script Name: script2

```
#!/bin/sh

echo “program name: $0”

echo “The number of arguments specified is $#”

echo The arguments are “$*”

grep “$1” $2

echo “finished”
```

Run the file with:

```
$ sh script2 bca student.lst
```

```
1114 | Samir P. Chowdhari | 12/02/1982 | Surat      | bcom | 8461626364
1114 | Samir P. Chowdhari | 12/02/1982 | Surat      | bcom | 8461626364
```

Following are the special parameters

Shell Parameter	Significance
$1,$2…	Stores first arguments, second argument of command line respectively
$#	Number of arguments specified in command line
$0	Name of executed command
$*	Complete set of positional parameter as a string
"$@"	Each quoted string treated as a separate argument (recommended over $*)
$?	Exit status of last command
$$	PID of the current shell
$!	PID of the last background job

exit and EXIT STATUS of Command

exit command is used with numeric argument. It is same as we use in c language.

exit 0 - If everything went fine use exit 0.

exit 1 - When something went wrong, use the exit 1.

$?

The parameter **$?** stores the exit status of last command.

```
$ grep bca student.lst ; echo $?
0                     It indicates success
$ grep bsc student.lst ; echo $?
1                     It indicates failure in finding pattern
$ grep bsc student.bst ; echo $?
grep: can't open student.bst
2                     It indicates failure in opening file
```

The Logical operator && and || (Conditional Execution)

The shell provide two operators that allow conditional execution – the && and || .

&& operator

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

Syntax

cmd1 && cmd2

Example:

```
$ grep  "bca" student.lst && echo "pattern found in file"
```

```
1111 | Amit B. Shah        | 01/01/1980 | Surat     | bca | 9429123423
1113 | Sunil C. Makwana    | 11/03/1981 | Vapi      | bca | 9925421213
1115 | Vimal K. Patel      | 18/03/1984 | Ahmedabad | bca | 9526126126
2234 | Jinal B. Chaudhri   | 17/07/1977 | Bharuch   | bca | 9427194271
3122 | Nilesh K. Makvana   | 25/03/1976 | Bharuch   | bca | 9712323456
pattern found in file
```

II operator

In || operator, The second command is executed only when the first fails.

Syntax

cmd1 || cmd2

Example:

```
$ grep  "bsc" student.lst && echo "pattern not found in file"

pattern not in file.
```

Conditional Statement

While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will explain following two decision making statements:

- The **if Conditional Statement**
- The **case...esac** statement

The if conditional statements:

If else statements are useful decision making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if statement:

- **if...fi statement**

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax:

```
if command1 is successfull
then
  execute commands
fi
```

Here Shell *Command1* is evaluated. If the Command1 is Successful then sequence of command inside the if...fi are executed.

if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

Syntax:

```
if  Command1 is successful
then
            execute commands
else
            execute commands
fi
```

Here Command1 is evaluated. If the command1 is successful then commands following if are executed. If Command1 is fail then commands of else are executed.

if...elif...else...fi statement

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax:

```
if  command1
then
  execute commands
elif command2
then
  execute commands
elif command3
then
  execute commands
else
  execute commands
fi
```

If command1 is successful then commands following if are executed.
If command2 is successful then the Commands following to command2 statement are executed.
….
…..
if all fails than commands following else are executed.

Example:
Script Name : script_if_fi

```
If cat test1
        echo “File found”
else
        Echo “File not found”
fi
```

test command

when you use if to evaluate expressions, you need the test statement because the true or false values returned by expressions can't directly handled by if. test uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by if for making decision.

test work in three ways:

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

Numeric comparision by test

The numeric comparison operators used by test always begin with a – (hypen) , followed by two-letter string and enclosed on either side by whitespace.

Numeric Comparison operator	Meaning
-eq	Equal
-ne	Not equal
-lt	Less than
-gt	Greater than
-le	Less than equal to
-ge	Greater than equal to

Some Example of Test

```
$ x=5; y=7; z=7.2
$ test $x –eq $y ; echo $?
1                              False (5 and 7 are not equal)
$ test $x –lt $y ; echo $?
0                              True
$ test $z –gt $y ; echo $?
1                              False ( 7.2 is not greater than 7 )
$ test $z –eq $y ; echo $?
0                              True (7.2 is equal to 7)
```

The last two tests prove that numeric comparison is restricted to integer only.

Scriptname : script_test_if

```
#!/bin/sh
a=10
b=20
if test $a –eq $b
then
        echo “ a and b are equal”
elif test $a –gt $b
then
        echo “a greater than b”
elif test $a –lt $b
then
        echo “b greater than a”
else
        echo “give proper input”
fi
```

String Comparison

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

Test	Meaning
s1=s2	True if String s1=s2
s1!=s2	True if String s1!=s2
-n str	String str is not a null string
-z str	String str is a null string
str	String str is assigned and not null
s1==s2	String s1=s2 (korn and bash only)

Script Name: Script_str_cmp

```
#!/bin/sh
echo “Pattern to be search \c”

read pname


If  [ -z “$pname” ] ; then
        echo “You have not entered the pattern”; exit 1
fi
```

```
echo  “Enter Filename \c”

read fname


If  [ ! -n “$fname” ] ; then
        echo “You have not entered the filename” ; exit 2
fi
grep “$pname” $fname
```

File test

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…)

```
$ test –f student.lst ; echo $?
0                                                   True, It is ordinary file
$ [ -x student.lst ] ; echo $?
1                                                   No, It is not executable file
```

Table: File related tests with test command or [ ]

Test	Meaning
-f file	True if file exists and is a regular file
-r file	True if file exists and is readable
-w file	True if file exists and is writable
-x file	True if file exists and is executable
-d file	True if file exists and is a directory
-s file	True if file exists and has a size greater than zero
-e file	True if file exists (Korn and Bash only)
-u file	True if file exists and has SUID bit set
-k file	True if file exists and has sticky bit set
-L file	True if file exists and is a symbolic link
f1 –nt f2	f1 is newer than f2 (Korn and Bash only)
f1 –ot f2	f1 is older than f2 (Korn and Bash only)
f1 –ef f2	f1 is linked to f2 (Korn and Bash only)

Script Name: script_file_test

```
#!/bin/sh
If [ ! –e $1 ] ; then
echo “File does not exist”
elif [ ! –r $1 ] ; then
        echo “File is not readable”
elif [ ! –w $1 ] ; then
        echo “File is both readable and writable”
fi
```

shorthand for test ([ ])

Short hand method for test command is [ ] . A pair of rectangular bracket ([]) enclosing the expression that evaluate by if command. you must provide white space around the operator, variable and inside [and].

Example of if…fi :

```
#!/bin/sh
a=10
b=20
if [ $a -eq $b ]
then
        echo "a is equal to b"
fi

if [ $a != $b ]
then
        echo "a is not equal to b"
fi
```

Output:

```
a is not equal to b
```

Example of if…elif…fi :

```
#!/bin/sh
a=10 ; b=20
if [ $a == $b ]
then
        echo "a is equal to b"
else
        echo "a is not equal to b"
fi
```

Output:

```
a is not equal to b
```

Example of if...elif...else...fi :

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

Output:

```
a is less than b
```

The case...esac Statement:

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

There is only one form of case...esac statement which is detailed here:

- case...esac statement

The basic syntax of the case...esac statement is to give an expression to evaluate and several different statements to execute based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

Syntax:

```
case expression/variable in
 pattern1)
   Statement(s) to be executed if pattern1 matches
   ;;
 pattern2)
   Statement(s) to be executed if pattern2 matches
   ;;
 pattern3)
   Statement(s) to be executed if pattern3 matches
   ;;
esac
```

Here the expression/variable is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example:

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
  "apple") echo "Apple pie is quite tasty."
  ;;
  "banana") echo "I like banana nut bread."
  ;;
  "kiwi") echo "New Zealand is famous for kiwi."
  ;;
esac
```

output:

```
New Zealand is famous for kiwi.
```

Looping (while)

while command repeatedly performs a set of instructions until condition becomes false.

Syntax

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

Example

Script Name: script_elist

```
ans=y
while [ $ans = “y” || $ans=”Y” ]
do
echo “Enter empcode  empname: \c”
read ecode ename
echo “$ecode|$ename” >> elist
echo “enter more (y/n)? \c”
read ans
done
```

Output

```
Enter empcode empname: 01 samir
Enter more(y/n)? y
Enter empcode empname: 02 rajesh
Enter more(y/n)? n

$ cat elist
01 samir
02 rajesh
```

Looping with a list (for)

for doesn’t test a condition, but uses a list.

Syntax

```
for variable in list

do

commands
```

done

Example

Script Name : script_5_no

```
for a in 1 2 3 4 5 ; do
echo $a
done
```

Output:

```
1
2
3
4
5
```

Script Name : script_lst_backup

```
for file in stud*
do
cp $file ${file}.bak
echo “$file is copied into $file.bak”
done
```

output

```
sh script_lst_backup
student.lst is copied into student.lst.bak
stud_city is copied into stud_city.bak
stud_ph is copied into stud_ph.bak

$ls *.bak
student.lst.bak        stud_city.bak        stud_ph.bak
```

Changing the file name extension (basename)

basename extracts the “base” filename from an absolute path.

```
$ basename /home/students/bca/student.lst
student.lst
```

When basename is used with two arguments, it strips(remove) off the second argument from the first argument.

```
$ basename abc.lst lst
abc.
```

Manipulating the positional parameters (set)

set command assigns its arguments to positional parameters $1, $2 and so on..
This feature is specially useful for picking up individual fields from the output of a program.

```
$ set 1111 1211 1323
$ echo “value of \$1 is $1, value of \$2 is $2, value of \$3 is $3”
value of $1 is 1111, value of $2 is 1211, value of $3 is 1323
$ echo “The $# arguments are $*”
The 3 arguments are 1111 1211 1323
```

We can use the set command to extract individual fields from the date output and without using cut.

```
$ set `date`
$ echo $*
Wed Jan 5 08:20:15 IST 2014
$ echo “The date today is $2 $3 $6”
The date today is Jan 5 2014
```

Shifting arguments left (shift)

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 first time $2 becomes $1, $3 becomes $2 and so on.

```
$ set `date`
$ echo $*
Wed Jan 5 08:20:15 IST 2014
$ echo $1 $2 $3
Wed Jan 5
$ shift
$ echo $1 $2 $3
Jan 5 08:20:15
$ shift 2
$ echo $1 $2 $3
08:20:15 IST 2014
```

The here document (<<)

here document is used to input the data at the command itself rather than using separate file.
Any command using standard input can also take input from the here document.

```
$ mailx Sharma << BODY
Your email given by you is printed
On `date`.
This is the report to you.
BODY
```

The here document start with symbol << and The three data lines given between the starting and ending tag BODY. It can be any name.
The below two command do the same thing.

```
$ sh script2 bca student.lst
OR
$ sh script2 <<DATA
>bca
>student.lst
>DATA
```

The above command display all records of bca.