

Unit 3. Shell Programming

- 3.1 Screen Editor "vi"
- 3.2 Environmental & user defined variables
- 3.3 Argument Processing(Y-positional)
- 3.4 Shell's interpretation at prompt-120(n)
- 3.5 Arithmetic expression evaluation(y-expr)
- 3.6 Control Structure(y)
- 3.7 Redirection(n)
- 3.8 Background process & priorities of process(n)
- 3.9 Conditional Execution(n-106)

Read statement

The read statement is used to make the shell interactive.
It is the input taking tool of the shell script.

Syntax: read var1 var2 var3

We can read values for multiple variables using a single read stmt.

If the number of input values given is more than the number of variables then the last values are assigned to the last variable.

If the number of input values given is less than the number of variables then the last variable is left unassigned.

e.g read n1 n2 n3

i/p 1 2 3 (n1=1,n2=2,n3=3)

1 2 (n1=1,n2=2 & n3 is left unassigned)

1 2 3 4 5 (n1=1,n2=2,n3=3 4 5)

3.3 Argument Processing /Using Command line arguments

In unix the command line arguments are provided by the positional parameters.

Positional parameters are the special variables that are assigned automatically when any arguments are passed at the command line while executing the script.

There are total 9 positional parameters (i.e \$1,\$2, \$3 \$9)

The first argument in the command line is assigned to \$1 the second to \$2 and so on upto \$9.

Some more special Parameters used by the shell

\$# : stores total number of arguments passed on command line

\$0(zero) : Name of the executed command

\$* : Complete set of positional parameter as a whole string

\$@ : Complete set of positional parameter as a whole string but the quoted arguments are treated as a separate argument

\$? : Exit status of the last command

If the value of exit status is 0 (zero); last command executed successfully

If the value of exit status is Non-zero value; the last command did not executed successfully.

\$\$: PID of current shel

#! : PID of last background job.

Explanation in class (not to be written in exam)

\$1=f2

\$2=f4

\$* =f2 f4

\$@ = f2 f4

"\$*"="f2 f4" (considering f2 f4 as single argument)

Cat "f2 f4" //file content of single file

"\$@" = "f2" "f4" (two arguments f2 f4)

Cat "f2" "f4" //two file content

Q-What is the difference between \$* and \$@?

Ans: \$* and \$@ stores the complete set of positional parameters as a single string. i.e. if the parameter set on the command line are:

\$ set a b c d e f

a will be stored in \$1, b will be stored in \$2 and so on...

The entire string a b c d e f will be stored in \$@ and \$*

i.e.

\$ echo \$*

o/p: a b c d e f

\$ echo \$@

o/p: a b c d e f

The only difference between \$* and \$@ is when \$* and \$@ are quoted. When \$@ is quoted it still considers the arguments as different parameters where as when \$* is quoted ; the entire line is considered as single argument.

```

[2020-08-07 23:11.26] ~
[lenovo.lenovo-PC] > set f1 f2

[2020-08-07 23:11.32] ~
[lenovo.lenovo-PC] > cat "$*"
cat: can't open 'f1 f2': No such file or directory

[2020-08-07 23:11.35] ~
[lenovo.lenovo-PC] > cat "$@"
this is the content of file f1
it is having two lines
This is content of file f2
this is second line
this is the third line

[2020-08-07 23:11.40] ~
[lenovo.lenovo-PC] > █

```

In the above figure set command is used to set *f1* and *f2* as command line arguments. *\$1* contains *f1* and *\$2* contains *f2* and *\$** and *\$@* contains *f1 f2*

If we try to print *"\$*"* using quotes; all the command lines arguments are considered as single string and hence it is trying to display the content of file *"f1 f2"* instead of *f1 f2*.

In the *"\$@"* case ;*f1* and *f2* are still two different arguments *f1* and *f2*. And hence it is able to display the content of file *f1* and *f2*.

The figure given below gives the interpretation of both *"\$*"* and *"\$@"*.

```

[2020-08-07 23:19.57] ~
[lenovo.lenovo-PC] > cat "f1 f2"
cat: can't open 'f1 f2': No such file or directory

[2020-08-07 23:20.06] ~
[lenovo.lenovo-PC] > cat "f1" "f2"
this is the content of file f1
it is having two lines
This is content of file f2
this is second line
this is the third line

[2020-08-07 23:20.12] ~
[lenovo.lenovo-PC] > █

```

The shift and set command

set: The set command is used to set the positional command explicitly. ie. When we pass arguments on command the positional parameters are automatically set. If we want to set the positional parameter explicitly the set command is used.

e.g

\$ set 123 345 567

\$_ (this command will set the 123 value to *\$1*, 345 to *\$2* and 567 to *\$3*)

It also sets the other parameters *\$#* and *\$**.

The command substitution is also used with this command for setting values for positional parameters.

```
Ksh for Windows --- Wipro UWIN Evaluation Licer
$ set `date`
$ echo $1
Thu
$ echo $2
Sep
$ echo $3
6
$ echo $4
15:53:17
$ echo $5
IST
$ date
Thu Sep 6 15:54:28 IST 2012
```

Value of \$1 is the first words of date command

Output of date command

This command will set the output of the date command to the positional parameters. i.e \$1 will be Thu, \$2 will be Sep, \$3 will be 6 and so on...

Note: set parses its arguments on the delimiters specified in the environment variable IFS, which by default is whitespace.

The set command is used with command substitution but in some cases it faces certain problems like, if the output of the command substitution is - as first argument then it generates error. Also if the output of the command substitution is null in that case also generates error.

e.g. (1) **set `ls -l u1`** (in the above case the first value of ls -l is - (which represents regular file)

set command interprets it as an option and finds to be bad one(i.e. not understandable)

(2) **set `grep ppp /etc/passwd`**

if the string ppp can't be located in the file, set will operate with no arguments and puzzles the user by displaying all the variables on the terminal.

The solution to both of these arguments is using "--" (two hyphens) after set command.

i.e. **set -- `ls -l u1`** and **set -- `grep ppp /etc/passwd`**

The set command does not treat the arguments following to -- as options and hence generates no error. The two hyphens also direct set command to suppress its default behavior if its arguments evaluate to null string.

shift:

Many scripts use the first arguments to indicate a separate entity. The other arguments could then represent a series of strings-the different patterns to be selected from a file. The values of the positional parameters are needed to be shifted. The shift command is used to perform the shifting operation.

e.g. In the above example the Thu value of date command is stored in \$1. if we don't need that value and want to start from the second argument in that case the shift command is used.

In the case of number of variables passed on the command line are more than 9, the shift command is used to accommodate all the command line arguments of the command line.

In this case if the first argument is to be used by the script than it has to be stored in variable before applying the shift command.

i.e. **\$prg1.sh f1.c f2.c f3.c f4.c f5.c f6.c f7.c f8.c f9.c f10.c (enter)**

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
\$1 \$2 \$3 \$4 \$5 \$6 \$7 \$8 \$9

now after applying shift command the values are shifted one step forward.

i.e. **\$shift**

now the \$1 will be f2.c, \$2 will be f3.c and \$9 will have value f10.c (which was not assigned earlier)

The f1.c is removed from \$1 and hence if it is required then it has to be stored before applying shift command

i.e.

\$var1=\$1

\$shift

Note: Every time the shift command is used the leftmost variable gets lost. If one has to start from the fourth parameter then one can apply **shift 3**

```
Ksh for Windows  --- Wipr
$ set 1 2 3 4 5
$ echo $*
1 2 3 4 5
$ shift 2
$ echo $*
3 4 5
```

1 is stored in \$1 , 2 is stored in \$2 and so on

after applying shift 2; 1 and 2 is removed and 3 is stored in \$1, 4 is stored in \$2 and so on.

The basename command

The basename is an external command. It extracts the base file name from the absolute pathname.

e.g. **\$basename /home/abc/pqr/f1.c**
f1.c

when the basename is used with two arguments, it strips off the second argument from the first argument:

\$basename f1.c c

f1. (the second argument c is stripped off from the first argument f1.c)

3.4 Shell's interpretation at prompt

3.5 Arithmetic expression evaluation

The expr statement

The expr is an external command which is used for computations in UNIX. It performs following features.

- Perform arithmetic operations on integers
- Manipulates strings.

Arithmetic Computations

expr can perform 4 basic arithmetic operations and also the modulus function.

i.e for two variables x any y the arithmetic operations can be:

Indicates the shell prompt.

```
$expr $x + $y
$expr $x - $y
```

Is required as * is considered as meta-character and \ will hide its special meaning

```
$expr $x \* $y
$expr $x / $y
$expr $x % $y
```

The above example shows the working of expr command on the command line. If the user wants to use the expr in shell scripts the user has to use it with command substitution.

e.g. to store the value of the summation of two values in a variable.

```
c=`expr $x + $y`
```

the value of sum is stored in variable c.

String Handling

The expr command is also used for handling strings. Using expr we can perform three basic string functions like,

✓ **Determine the length of the string:**

The length of the string is counted using the regular expression `".*"`. This regular expression signifies that the expr has to print the number of characters matching the pattern.

e.g.

```
$ str="Unix"
$ expr "$str" : '.*'
4
```

Also we can use it in the shell programming like,

```
str=Shellprogramming
```

```
l=`expr $str : '.*'`
```

echo \$1 (the output for the above command will be 16)

✓ **Extract the substring**

The expr command can extract a string enclosed by the escaped characters `\(` and `\)`. If the user wants to extract 2 digits year from a 4 digit string it should be done in the following manner.

Eg. (1) `$ str=2003`

```
$expr "$str" : '..\(..\)'
```

03 (the last two characters are extracted from the full year 2003)

(2) `$filename=prg1.c`

```
$expr "$filename" : '\(..*\)\.c'
```

prg1 (the .c part of the string is removed or rather only the name of the file is extracted)

✓ **Locate the position of a character in the string**

The expr command 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 str

variable , the user has to count the number of characters which are not d ([^d]*), followed by a d.

e.g. `$str=abcde ; expr "$str" : '[^d]*d'`

4 (the output is the first occurrence of the character d in the given string "abcde")

3.6 Control Structure

Using test and [] to evaluate expressions

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 statement for decision making

Test works in three ways:

- Compare two numbers
- Compare two strings
- Checks file's attributes

Numeric comparison:

The comparison operators in test are as given below:

-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

The comparison syntax is: **test var1 operator var2**

e.g. `test $x -eq $y`

Note: the numeric comparison is restricted to the integers only.

We can use **test** with the **if statement** for testing condition.

e.g.

```
if test $# -eq 0
then
    echo "no arguments are given on the command line"
else
    echo "some arguments are given on the command line"
fi
```

Also `[]` is used for testing the condition just like the test statement is used.

test \$x -eq \$y or

[\$x -eq \$y]

i.e. if we rewrite the above script using `[]` it can be written as following:

```
if [ $# -eq 0 ]
then
    echo "no arguments are given on the command line"
else
    echo "some arguments are given on the command line"
fi
```

String Comparison

test can be used to compare two strings with another set of operators.

i.e.

`str1=str2` operator for checking equality

`str1!=str2` operator for checking inequality

`-n string` is used to check the string is not a null string

`-z string` is used to check the string is a null string

`Str1==str2` is used to check str1 and str2 are same or not. (this is used only in Korn and Bash)

test also permits to check more than one permission at a same time using `"-a"` (AND) and `"-o"` (OR) operator.

e.g.

```
if [ -n "$str1" -a -n "$str2" ]
then
    echo "both the string are not null strings"
else
    echo "anyone of the string is null string"
fi
```

Checking file attributes

test can be used to check various file attributes also, like file is readonly, writeable, regular, directory etc.

e.g **test -f emp.lst** or **[-f emp.lst]** : will check whether the file is ordinary file or not.

The various file argument tests used with test command are:

| | |
|------------------------|--|
| <code>-f file</code> | File exists and is a regular file |
| <code>-r file</code> | File exists and is a readable |
| <code>-w file</code> | File exists and is a writeable |
| <code>-x file</code> | File exists and is a executable |
| <code>-file</code> | File exists and is a directory |
| <code>-s file</code> | File exists and has a size greater than zero |
| <code>-e file</code> | File exists(Korn and Bash only) |
| <code>-u file</code> | File exists and has SUID bit set |
| <code>-k file</code> | File exists and has sticky bit set |
| <code>-L file</code> | File exists and has symbolic link (Korn and Bash only) |
| <code>F1 -nt f2</code> | F1 is newer than f2(Korn and Bash only) |
| <code>F1 -ot f2</code> | F1 is older than f2(Korn and Bash only) |
| <code>F1 -ef f2</code> | F1 is linked to f2(Korn and Bash only) |

```
e.g.    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 not writable"
else
    echo " file is both readable and writeable"
fi

```

Logical Operators:

Logical operators supported by Unix are:

- -a (Logical AND)
- -o (Logical OR)
- -! (Logical NOT)

Control structure:

Control structure alters the follow of execution of the shell script. Unix supports two types of control structure.

- Decision making
- Looping

Decision making control structure are:

1. If statement
2. Case-esac statement

The if conditional statement:

Syntax:

| | |
|---|--|
| <pre> if command/condition then stmt1 //true block else stmt2 //false fi </pre> | <pre> if command/condition then stmt1 elif command/condition then stmt2 else stmt3 fi </pre> |
| <pre> if command/condition then stmt1 fi </pre> | |

e.g.

```

if grep "surat" emp // read the exit status $?-0 (true) -non zero value (false)
then
    echo "pattern found"
else
    echo "pattern not found"

```

The case Conditional

The case structure is used as a conditional statement in the shell.

The statement matches an expression for more than one alternative, and uses a compact construct to allow multiway branching. It also handles the string tests.

Syntax

case expression in

```
pattern1) commands1 ;; // ;; signifies the break;
pattern2) commands2 ;;
pattern3) commands3 ;;
```

.....

esac

e.g.

```
echo "Menu \n
```

1. List of files \n 2.Process of the user \n 3.Today's date \n 4. Quit UNIX \n Enter the appropriate option : \c"

```
read ch //scanf()
```

```
case "$ch" in
```

```
1) ls -l ;;
2) ps -f ;;
3) date ;;
4) exit ;;
*) echo "Invalid choice"
```

```
esac
```

- case can't handle relational and file test, but it matches strings with compact code. It works effectively when the string is fetched using command substitution.
- case can also handle numbers but only by treating them as strings. Numeric checks like `$n -gt 0` is not possible with case.

Matching multiple patterns using case

case can be used for checking multiple patterns. The symbol "|" is used to delimit the multiple patterns. i.e `y|Y` can be used to match with `y` and `Y` any of the character.

```
echo "Do you wish to continue?(y/n): \c"
```

```
read answer
```

```
case "$answer" in
```

```
y|Y) ;★———— if y or Y is given as a value to the answer variable than no  
action will be performed
```

```
n|N) exit ;★———— if n or N is given as a value to the answer variable  
than will exit from the program
```

```
esac
```

Use of wild card characters with case

The user can use meta-characters for matching patterns in case statement. i.e. the characters like `*`, `[]` are used with the pattern matching in the case.

e.g.

```
echo "Do you wish to continue?(y/n): \c"
```

```
read answer
```

```
case "$answer" in
```

```
[yY][eE]*) ★:———— will match the pattern YES, yes, Yes, yES ...etc
```

```
Ni exit ;; ←———— will match the pattern NO, no, No and nO
```

```
*) echo "Invalid response"
```

esac

No,NO,nO,no

The wild card characters are considered as special characters and their special meaning are preserved while matching the patterns in case statement.

The while: looping

UNIX supports three looping structure: while, until and for.

The while statement repeatedly performs a set of instructions until the control command returns a true exit status.

The syntax for the while loop is:

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

The commands enclosed by **do** and **done** are executed repeatedly as long as condition remains true. The various unix commands and test command can be used as condition.

e.g.

```
ans=y
while [ $ans = "y" ]
do
    echo "Enter the value of code and description"
    read code des
    echo "$code| $des" >> newfile
    echo "Do you want to continue"
    read ans
    case $ans in
        y* | Y*) ans=y ;;
        n * | N*) ans=n ;;
        *) ans=y ;;
    esac
done
```

The user can send the output of the whole while loop in any file by using the redirection operator with the keyword done itself.

```
done > newfile
```

Using the about statement the whole output will be sent to the file.

This can be used with the **fi** and **esac** also.

e.g. done < param.lst (statements in loop take input from param.lst)

done | while true (Pipes output to while loop)

fi > foo (affects statements between if and fi)

esac > foo (affects statements between case and esac)

Using sleep command in while loop

The sleep command is used to keep the loop waiting for some time.

e.g. while [! -r invoice.lst]

```
do
  sleep 60
done
```

This script will check whether the file exists or not if it does not exist the system will sleep for 1 minute and again will check about the existence of the file.

Infinite loop

If in a loop the condition is set as it always evaluates to true the loop becomes an infinite loop.

```
e.g. while true
do
  command
done
```

Such loops can be used to perform specific task that has to be done infinitely in the system e.g. checking the number of users after every 1 hour and so on. The loop has to be sent to background as the shell will not be able to execute any other command if the infinite loop is running in foreground. If the loop is sent to background and the user wants to stop the execution of the loop, the user has to kill the loop using kill command.

The for loop

The **for** and **until** loop in unix does not test a condition instead they work on a list.

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

The loop body uses the keyword do and done. The values in the list are one by one assigned to the variable. Each whitespace separated word in list is assigned to variable in turn and commands are executed until the list is exhausted.

```
e.g.
for fname in cat *
do
  cp $fname ${fname}.bak
  echo $fname copied to $fname.bak
done
```

The above loop comprises with the name of all the files in the current directory.

The possible sources of the list can be as given below:

The list from the variables: The series of variables in the command line. They are evaluated by the shell before executing the loop.

e.g. for var in \$PATH \$HOME \$MAIL

The list from command substitution: The command substitution can also be used as a list.

e.g. for file in `cat clist`

The list from wild cards: The list can also comprise of wild cards. The shell interprets them as filenames.

e.g. for file in *.htm *.html

This loop works on every HTML file in the current directory.

The list from positional parameters: for uses to process positional parameters that are assigned from command line arguments.

e.g. for pattern in "\$@"

The until loop

The until loop works just like the while loop, except that it loops as long as the exit status of the command is false. It can be called as complement of while loop.

syntax:

```
until command
do
    command
done
```

3.7 Redirection

3.8 Background process & priorities of process

3.9 Conditional Execution

The logical Operators && and || -Conditional execution

The shell provides two conditional operators

&& and ||

Syntax : cmd1 && cmd2

cmd1 || cmd2

&& : delimits two commands i.e. cmd2 is executed only when cmd1 succeeds

e.g. \$ grep 'surat' emp && echo "pattern found"

|| : in this the second command is executed only when the first command fails.

e.g. . \$ grep 'surat' emp || echo "pattern not found"

This command works well with simple conditions for complex decision making if statement is used.

Cmd1 && cmd2

Fail -> cmd2 will not be executed

Succ -> cmd2 will be executed

Cat f1 && echo "file found"

Cmd1 || cmd2

Fail-> cmd2 will be executed

Succ-> cmd2 will not be executed

(1) **Command1 && command2 || command3**

| | | |
|-------------|-------------|-------------|
| Cmd1 | Cmd2 | Cmd3 |
|-------------|-------------|-------------|

| | | |
|----------------|---------------------|--------------------|
| Fails | Not executed | Execute |
| Success | Success | Not Execute |
| Success | Fails | Execute |

Command1-> fail-> cmd2 not execute (cmd3 will execute)

_>succ -> cmd2 will execute-> fail -> cmd3 will execute

-> succ-> cmd3 will not execute

Cat f1 && cat f2|| echo "file found"

```
[2020-07-29 10:42.53] ~
[lenovo.lenovo-PC] > cat jvp && cat jvp1 || echo "file not found"
echo "Hello"
this is going to be written in filew
new line
jinal is my
name

[2020-07-29 10:43.27] ~
[lenovo.lenovo-PC] > cat jvp && cat jin || echo "file second not found"
echo "Hello"
cat: can't open 'jin': No such file or directory
file second not found

[2020-07-29 10:44.02] ~
[lenovo.lenovo-PC] > cat jin && cat jin || echo "file second not found"
cat: can't open 'jin': No such file or directory
file second not found
```

(2) Command1 || command2 && command3

| | | |
|----------------|---------------------|---------------------|
| Cmd1 | Cmd2 | Cmd3 |
| Fails | Fails | Not executed |
| Fails | Successful | Execute |
| Success | Not executed | Execute |

Shell Scripts

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

The script are generally created with the **.sh** extension

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 belongs to the same type.)

UNIX provides facility of **Interpreter line** in tshe first line of the script to specify a different shell for the script.

Interpreter line

The first line of the shell script can be considered as interpreter line.

It starts with **#!** and then is followed by the path name of the shell to be used for running the script.

e.g.

```
#!/bin/sh
# shell Script to print today's date and current shell
echo "Today's date is : `date`"
echo "This month's calendar:"
cal `date "+%m 20%y"`
echo "My shell : $SHELL"
```

Running the script

Syntax: **\$ sh name_of_the_script**

(explicitly spawn the child of the users choice with the script name as argument)

or

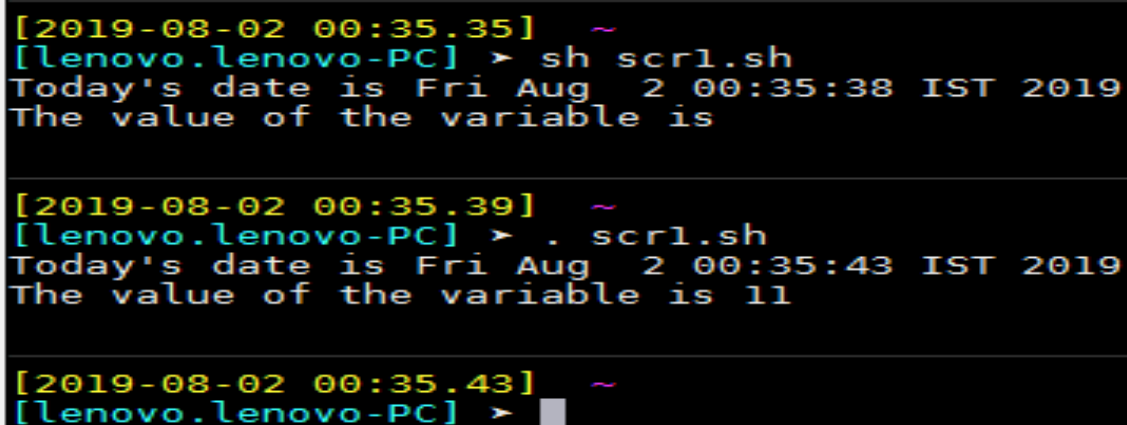
\$ name_of_the_script

When the sh is used the interpreter line is ignored. If the script is run without "sh" then the PATH variable has to be set. The path that contains the script has to be added to the PATH variable.

The another approach to execute the script is

\$.scriptname

Using this approach the script is run in the current shell. It can be verified by checking the value of local variable.



```
[2019-08-02 00:35.35] ~
[lenovo.lenovo-PC] > sh scr1.sh
Today's date is Fri Aug  2 00:35:38 IST 2019
The value of the variable is

[2019-08-02 00:35.39] ~
[lenovo.lenovo-PC] > . scr1.sh
Today's date is Fri Aug  2 00:35:43 IST 2019
The value of the variable is 11

[2019-08-02 00:35.43] ~
[lenovo.lenovo-PC] > █
```

The Here document

The is read in a script in some cases the user needs to read data from the script itself rather than reading from the file. This is done with the here document (<<). The user can

use any marker to start and end the input. The starting and ending marker string should be same.

e.g. mailx Sharma << msg1

this is the demonstration of the command here document

msg1

In the above example the here document symbol is followed by a single line and a delimiter (the string msg1). The line is treated as an input to the command mailx. The advantage of using HERE Document is that the command doesn't have to read the external file the input is given with the command itself.

The here document can also be used with the shell scripts.

e.g for a script f1.sh

```
echo "Enter the name of the file"
read fname
echo "Enter the pattern to be searched"
read pname
grep $pname $fname
```

The here document can be used as given below:

\$sh f1.sh << END

> emp.lst

> surat

END

The input for the script will be given using here document. Using the here document the interactive scripts will be made non interactive.

Debugging Shell scripts

The shell scripts can be debugged using the set command.

The set command is used with the -x option for debugging. When used inside the script or even at the prompt, it echoes each statement on the terminal, preceded by a + as it is executed. To debug the script the statement "**set -x**" has to be written as the first statement of the script.

set +x => turns the debugging off

set -x => turns the debugging on