UNIT 5.
Shell Programming

Structure

5.0 Introduction
5.1 Objectives
5.2 Why Shell Programming?
5.3 When not to use shell scripts
5.4 Starting Off With a Sha-Bang
5.5 Redirecting I/O
    5.5.1 Standard Imput and Standard Output
        5.5.1.1 putting text in a file
        5.5.1.2 understanding access permissions
    5.5.2 pipes and filters
5.6 Invoking the script
5.7 Special Characters
5.8 Introduction to Variables and Parameters
    5.8.1 Variable Assignment
    5.8.2 Positional Parameters
    5.8.3 Special meanings of escaped characters
    5.8.4 Examples of using pipelines to connect commands
5.9 If Then Else Constructs
    5.9.1 File test operators
    5.9.2 Comparison operators (binary)
    5.9.3 Internal Variables
5.10 Loops and Branches
    5.10.1 Loops
        5.10.1.1 for loop
        5.10.1.2 while loop
        5.10.1.3 until loop
5.11 Nested Loops
5.12 Branching Construct
5.13 Examples of Bourne shell scripts
    5.13.1 To read i/p to a command and process it in some way
    5.13.2 To read commands from the terminal and process them
5.14 Summary
5.15 Check Your Progress

5.0 Introduction

The shell is a command interpreter. More than just the insulating layer between the operating system kernel and the user, it's also a fairly powerful programming language. A shell program, called a *script*, is an easy-to-use tool for building applications by "gluing" together system calls, tools, utilities, and compiled binaries. Virtually the entire repertoire of **LINUX** commands, utilities, and tools is available for invocation by a shell script. If that were not enough, internal shell commands, such as testing and loop constructs, give additional power and flexibility to scripts. Shell scripts lend themselves

exceptionally well to do administrative system tasks and other routine repetitive jobs not requiring the bells and whistles of a full-blown tightly structured programming language.

The shell is just an ordinary program that can be called by a UNIX command. However, it contains some features (like variables, control structures, and so on) that make it similar to a programming language. You can save a series of shell commands in a file, called a *shell script*, to accomplish specialized functions.

5.1 Objectives

At the end of this unit, You would be able to

- Undestand why shell programming is necessary
- Know when not to use shell scripts
- How to redirect i/o to & from a file
- Differentiate among pipes & filters
- Explain different control constructs
- Write own shell script programs

5.2 Why Shell Programming?

A working knowledge of shell scripting is essential to everyone wishing to become reasonably adept at system administration, even if they do not anticipate ever having to actually write a script. Consider that as a Linux machine boots up, it executes the shell scripts in `/etc/rc.d` to restore the system configuration and set up services. A detailed understanding of these startup scripts is important for analyzing the behavior of a system, and possibly modifying it.

Writing shell scripts is not hard to learn, since the scripts can be built in bite-sized sections and there is only a fairly small set of shell-specific operators and options to learn. The syntax is simple and straightforward.

A shell script is a "quick and dirty" method of prototyping a complex application. Getting even a limited subset of the functionality to work in a shell script, even if slowly, is often a useful first stage in project development. This way, the structure of the application can be tested and played with, and the major pitfalls found before proceeding to the final coding in C, C++, Java, or Perl.

5.3 When not to use shell scripts

- resource-intensive tasks, especially where speed is a factor (sorting, hashing, etc.)
- procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use C++ or FORTRAN instead)
- cross-platform portability required (use C instead)
- complex applications, where structured programming is a necessity (need typechecking of variables, function prototypes, etc.)
- situations where security is important, where you need to guarantee the integrity of your system and protect against intrusion, cracking, and vandalism

- project consists of subcomponents with interlocking dependencies
- extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion)
- need multi-dimensional arrays
- need data structures, such as linked lists or trees
- need to generate or manipulate graphics or GUIs
- need direct access to system hardware
- need port or socket I/O
- need to use libraries or interface with legacy code
- proprietary, closed-source applications (shell scripts are necessarily Open Source)

If any of the above applies, consider a more powerful scripting language, perhaps Perl, Tcl, Python, or possibly a high-level compiled language such as C, C++, or Java.

Bash, an acronym for **"Bourne-Again Shell"** . Bash has become a *de facto* standard for shell scripting on all flavors of LINUX.

5.4 Starting Off With a Sha-Bang

In the simplest case, a script is nothing more than a list of system commands stored in a file. At the very least, this saves the effort of retyping that particular sequence of commands each time it is invoked.

Example : cleanup: A script to clean up the log files in /var/log

```
# cleanup
# Run as root, of course.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Logs cleaned up."
```

There is nothing unusual here, just a set of commands that could just as easily be invoked one by one from the command line on the console. The advantages of placing the commands in a script go beyond not having to retype them time and again. The script can easily be modified, customized, or generalized for a particular application.

The *sha-bang* (#!) at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated. The #! is actually a two-byte "magic number", a special marker that designates a file type, or in this case an executable shell script. Immediately following the *sha-bang* is a path name. This is the path to the program that interprets the commands in the script, whether it be a shell, a programming language, or a utility. This command interpreter then executes the commands in the script, starting at the top (line 1 of the script), ignoring comments.

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
```

Note that the path given at the "sha-bang" must be correct, otherwise an error message, usually "Command not found" will be the only result of running the script.

Example of a simple shell script:

```
cat display
 # This script displays the date, time, username and current directory.
   echo "Date and time is:"
   date
   echo
   echo "Your username is: `whoami` \\n"
   echo "Your current directory is: \\c"
   pwd
```

The first line beginning with a hash (#) are comments and are not interpreted by the shell. The backquotes (`) around the command **whoami** illustrate the use of command substitution.

The `\\n` is an option of the echo command that tells the shell to add an extra carriage return at the end of the line. The `\\c` tells the shell to stay on the same line.

To make a file containing a shell script executable and then run the script:

```
display
display: execute permission denied
chmod u+x display
display
Date and time is:
Mon Mar  8 10:51:17 GMT 1993
Your username is: erpl08
Your current directory is: /home/erpl08/scripts
```

5.5 Redirecting I/O

Whether or not they use them, all LINUX programs have the following:

- Standard input (stdin)
- Standard output (stdio)
- Standard error output (stderr)

By default:

- Standard input connected to user's keyboard
- Standard output connected to user's display
- Standard error output connected to user's display

The angle bracket character changes this:

- < (comes from) redirects the standard input
- > (goes to) redirects the standard output

5.5.1 Standard Input and Standard Output

Many LINUX commands read input (such as a file) and write output.

In general, if no filename is specified in a command, the shell takes whatever you type on your keyboard as input to the command (after you press the first [RETURN] to start the command running, that is). Your terminal keyboard is the command's *standard input*.

As a command runs, the results are usually displayed on your terminal screen. The terminal screen is the command's *standard output*.

So, by default, each command takes its input from the standard input and sends the results to the standard output.

These two default cases of input/output can be varied. This is called *input/output redirection*. You can use a given file as input to a command that doesn't normally accept filenames by using the "<" (less-than symbol) operator. For example, the following command mails the contents of the file *hell.txt* to *bigboss@corp*:

```
% mail rao@rediffmail.com < hello.txt
%
```

You can also write the results of a command to a named file or some other device instead of displaying output on the screen using the > (greater-than symbol) operator. The pipe operator | sends the standard output of one command to the standard input of another command. Input/output redirection is one of the nicest features of LINUX because of its tremendous power and flexibility.

5.5.1.1 Putting Text in a File

Instead of always letting the output of a command come to the screen, you can redirect output into a file. This is useful when you have a lot of output that would be hard to read on the screen or when you put files together to create a bigger file.

As we've seen, the **cat** command can display a short file. It can also be used to put text into a file, or to create a bigger file out of smaller files.

5.5.1.1 The > operator

When you add "> *filename*" to the end of a command line, the results of the command are diverted from the standard output to the named file. The > symbol is called the *output redirection operator*.

For example, let's use **cat** with this operator. The contents of the file that you'd normally see on the screen (from the standard output) are diverted into another file:

```
% cat /etc/passwd > password
% cat password
root::0:0:Root:/:/bin/sh
daemon:NONE:1:1:Admin:/:
  .
```

```
  .
  .
john::128:50:John Doe:/usr/john:/bin/sh
%
```

You can use the > redirection operator with any command that sends text to its standard output - not just with **cat**. For example:

```
% who > users
```

We've sent the output of **who** to a file called *users* and the output of **date** to the file named *today*. Listing the directory shows the two new files. Let's look at the output from the **who** and **date** commands, regarding these two files:

```
% cat users
tim     tty1    Aug 12  07:30
john    tty4    Aug 12  08:26
```

You can also use the **cat** command and the > operator to make a small text file. We told you earlier to type `[CTRL-D]` if you accidentally enter **cat** without a filename. This is because the **cat** command alone takes whatever you type on the keyboard as input. Thus, the command:

cat > *filename*

takes input from the keyboard and redirects it to a file. Try the following example:

```
% cat > check.txt
Finish report by noon
Lunch with Xannie
Swim at 5:30
^D
%
```

cat takes the text that you typed as input, and the > operator redirects it to a file called *check.txt*. Type `[CTRL-D]` on a new line by itself to signal the end of the text. You should get a shell prompt.

You can also create a bigger file out of many smaller files using the **cat** command and the > operator. The form:

cat *file1 file2* > *newfile*

creates a file *newfile*, consisting of *file1* followed by *file2*.

> **NOTE:** If you are using the > (output redirection) operator, you should be careful not to overwrite the contents of a file accidentally. Your system may let you redirect output to an existing file. If so, the old file will be deleted (or, in LINUX lingo, "clobbered"). Be careful not to overwrite a much-needed file! Many shells can protect you from this risk. In the C shell, use the command **set noclobber**. The Korn shell and **bash** command is **set -o noclobber**. Enter the command at a shell prompt or put it in your shell's startup file After that, the shell will not allow you to redirect onto an existing file and overwrite its contents.

This doesn't protect against overwriting by LINUX commands like **cp**; it works only with the > redirection operator. For more protection, you can set LINUX file access permissions.

5.1.1.2 The >> operator

You can add more text to the end of an existing file, instead of replacing its contents, by using the >> (append redirection) operator. Use it like the > (output redirection) operator. So,

cat *file2* >> *file1*

appends the contents of *file2* to the end of *file1*. For an example, let's append the contents of the file *users*, and also the current date and time, to the file *diary*. Then we display the file:

```
% cat users >> diary
% cat diary
Tue Aug 12 08:36:09 EDT 1997
Finish report by noon
Lunch with Xannie
Swim at 5:30
tim     tty1    Aug 12  07:30
john    tty4    Aug 12  08:26
Tue Aug 12 09:07:24 EDT 1997
%
```

5.5.1.2 Understanding access permissions

There are three types of permissions:

```
r - 1  read the file or directory
w - 2 write to the file or directory
x - 4 execute the file or search the directory
```

Each of these permissions can be set for any one of three types of user:

```
u  the user who owns the file (usually you)
g  members of the group to which the owner belongs
o  all other users
```

The access permissions for all three types of user can be given as a string of nine characters:

```
  user    group   others
 r w x    r w x    r w x
```

5.5.2 Pipes and Filters

In addition to redirecting input/output to a named file, you can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a ***pipe***. To make a pipe, put a vertical bar (|) on the command line between two commands. When a pipe is set up between two commands, the standard output of the command to the left of the pipe

symbol becomes the standard input of the command to the right of the pipe symbol. Any two programs can form a pipe as long as the first program writes to standard output and the second program reads from standard input.

When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output (which may be piped to yet another program), it is referred to as a ***filter***. One of the most common uses of filters is to modify output. Just as a common filter culls unwanted items, the LINUX filters can be used to restructure output.

Almost all LINUX commands can be used to form pipes. Some programs that are commonly used as filters are described below. Note that these programs aren't used only as filters or parts of pipes. They're also useful on their own.

For example:
grep

The **grep** program searches a file or files for lines that have a certain pattern. The syntax is:

grep *pattern file(s)*

The name "grep" derives from the **ed** (a LINUX line editor) command **g/re/p** which means "*g*lobally search for a *r*egular *e*xpression and *p*rint all lines containing it." A *regular expression* is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of **grep** is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give **grep** a filename to read, it reads its standard input; that's the way all filter programs work:

```
% ls -l | grep "Aug"
-rw-rw-rw-   1 rao  doc     11008 Aug  6 14:10 ch02
%
```

First, our example runs **ls -l** to list your directory. The standard output of **ls -l** is piped to **grep**, which only outputs lines that contain the string "Aug" (that is, files that were last modified in August). Because the standard output of **grep** isn't redirected, those lines go to the terminal screen.

grep options let you modify the search. Table 5.1 lists some of the options.

Table 5.1: Some grep Options

Option	Description
-v	Print all lines that do not match pattern.
-n	Print the matched line and its line number.
-l	Print only the names of files with matching lines (letter "l").
-c	Print only the count of matching lines.
-i	Match either upper- or lowercase.

Next, let's use a regular expression that tells **grep** to find lines with "carol", followed by zero or more other characters (abbreviated in a regular expression as ".*"), then followed by "Aug".

```
% ls -l | grep "carol.*Aug"
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
%
```

sort

The **sort** command arranges lines of text alphabetically or numerically. The example below sorts the lines in the *food* file alphabetically. sort doesn't modify the file itself; it reads the file and writes the sorted text to the standard output.

```
% sort food
Bangkok Wok
Big Apple Deli
Isle of Java
Sweet Tooth
```

sort arranges lines of text alphabetically by default. There are many options that control the sorting. Some of these are given in Table 5.2.

Table 5.2: Some sort Options

Option	**Description**
-n	Sort numerically (example: 10 will sort after 2), ignore blanks and tabs.
-r	Reverse the order of sort.
-f	Sort upper- and lowercase together.
+*x*	Ignore first *x* fields when sorting.

More than two commands may be linked up into a pipe. Taking a previous pipe example using **grep**, we can further sort the files modified in August by order of size. The following pipe consists of the commands **ls**, **grep**, and **sort**:

```
% ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
%
```

This pipe sorts all files in your directory modified in August by order of size, and prints them to the terminal screen. The **sort** option **+4n** skips four fields (fields are separated by blanks) then sorts the lines in numeric order. So, the output of **ls** (actually, the output of **grep**) is sorted by the file size (the fifth column, starting with 1605). Both **grep** and **sort** are used here as filters to modify the output of the **ls -l** command. If you wanted to email this listing to someone, you could add a final pipe to the **mail** command. Or you could print the listing by piping the **sort** output to your printer command (like **lp** or **lpr**).

5.6 Invoking the script

Having written the script, you can invoke it by **`sh scriptname`**, or alternately **`bash scriptname`**. Much more convenient is to make the script itself directly executable with a chmod.

Either:
`chmod 555 scriptname` (gives everyone read/execute permission)

or
chmod +rx scriptname (gives everyone read/execute permission)

chmod u+rx scriptname (gives only the script owner read/execute permission)

Having made the script executable, you may now test it by **./scriptname**. If it begins with a "sha-bang" line, invoking the script calls the correct command interpreter to run it.

As a final step, after testing and debugging, you would likely want to move it to `/usr/local/bin` (as root, of course), to make the script available to yourself and all other users as a system-wide executable. The script could then be invoked by simply typing **scriptname [ENTER]** from the command line.

5.7 Special Characters

Special Characters Found In Scripts.

\# **Comments:** Lines beginning with a # (with the exception of #) are comments.

```
# This line is a comment.
```

; **Command separator:** [Semicolon] Permits putting two or more commands on the same line.

```
echo hello; echo there
```

;; **Terminator in a case option:** [Double semicolon]

```
case "$variable" in
abc)  echo "$variable = abc" ;;
xyz)  echo "$variable = xyz" ;;
esac
```

! **reverse (or negate) the sense of a test or exit status:** The ! operator inverts the exit status of the command to which it is applied (see Example 3.2). It also inverts the meaning of a test operator. This can, for example, change the sense of "equal" (=) to "not-equal" (!=). The ! operator is a Bash keyword.

In a different context, the ! also appears in indirect variable references.

* **wild card:** [asterisk] The * character serves as a "wild card" for filename expansion in globbing, as well as representing any number (or zero) characters in a regular expression.

A double asterisk, **, is the exponentiation operator.

& **Run job in background:** A command followed by an & will run in the background.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

~ home directory: [tilde] This corresponds to the $HOME internal variable. *~bozo* is bozo's home directory, and **ls ~bozo** lists the contents of it. ~/ is the current user's home directory, and **ls ~/** lists the contents of it.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~nonexistent-user
~nonexistent-user
```

5.8 Introduction to Variables and Parameters

Variables are at the heart of every programming and scripting language. They appear in arithmetic operations and manipulation of quantities, string parsing. A variable is nothing more than a location or set of locations in computer memory.

Example : Variable assignment and substitution

```
# Variables: assignment and substitution

#  If "VARIABLE =value",
#+ script tries to run "VARIABLE" command with one argument,
"=value".

a=375
hello=$a

# No space permitted on either side of = sign when initializing
variables.
#  It is permissible to set multiple variables on the same line,
#+ if separated by white space.

var1=variable1  var2=variable2  var3=variable3
echo   “welcome to LINUX LAB”
echo "var1=$var1   var2=$var2  var3=$var3"
```

5.8.1 Variable Assignment

= the assignment operator (*no space before & after*)

Example : Plain Variable Assignment

```
1)
# When is a variable "naked", i.e., lacking the '$' in front?
# When it is being assigned, rather than referenced.
```

```
# Assignment
a=879
echo "The value of a is $a"

2)
# Assignment using 'let'
let a=16+5
echo "The value of a is now $a"

3)
# In a 'for' loop (really, a type of disguised assignment)
echo -n "The values of a in the loop are "
for a in 7 8 9 11
do
  echo -n "$a "
done

4)
# In a 'read' statement (also a type of assignment)
echo -n "Enter a "
read a
echo "The value of a is now $a"
exit 0
```

Example : Variable Assignment, plain and fancy

```
1)
a=23              # Simple case
echo $a
b=$a
echo $b

# Now, getting a little bit fancier...
2)
a=`echo Hello!`   # Assigns result of 'echo' command to 'a'
echo $a
3)
a=`ls -l`         # Assigns result of 'ls -l' command to 'a'
echo $a

exit 0
```

5.8.2 Positional Parameters

Arguments passed to the script from the command line - $0, $1, $2, $3... $0 is the name of the script itself, $1 is the first argument, $2 the second, $3 the third, and so forth. [1] After $9, the arguments must be enclosed in brackets, for example, ${10}, ${11}, ${12}.

The **shift** command reassigns the positional parameters, in effect shifting them to the left one notch.

`$1` <--- `$2`, `$2` <--- `$3`, `$3` <--- `$4`, etc.

The old `$1` disappears, but `$0` does not change. If you use a large number of positional parameters to a script, **shift** lets you access those past `10`, although {bracket} notation also permits this.

Example 5-7. Using shift

```
# Using 'shift' to step through all the positional parameters.

# Name this script something like shft,
# and invoke it with some parameters, for example
# ./shft a b c def 23 skidoo

until [ -z "$1" ]  # Until all parameters used up...
do
  echo -n "$1 "
  shift
done
```

5.8.3 Special meanings of escaped characters

These are used with **echo** and **sed**

\n means newline
\r means return
\t means tab
\v means vertical tab
\b means backspace
\a means "alert" (beep or flash)
\0xx translates to the octal ASCII equivalent of *0xx*

5.8.4 Examples of using pipelines to connect commands

To pipe the output from one command into another command:--

```
who | wc -l
342
```

This command tells you how many users are currently logged in on the system.

The standard output from the **who** command - a list of all the users currently logged in on the system - is piped into the **wc** command as its standard input. Used with the `-l` option this command counts the numbers of lines in the standard input and displays the result on the standard output.

To connect several commands together:--

```
ps -aux|grep rao|sort +5 -6|less
```

The first command `ps -aux` outputs information about the processes currently running. This information becomes the input to the command `grep rao` which searches for any lines that contain the username "rao". The output from this command is then sorted on the sixth field of each line, with the output being displayed one screenful at a time using the `less` pager.

5.9 If Then Else Constructs

Every reasonably complete programming language can test for a condition, then act according to the result of the test.

- An **if/then** construct tests whether the exit status of a list of commands is 0, and if so, executes one or more commands.
- There exists a dedicated command called **[** (left bracket special character). It is a synonym for **test**, and a builtin for efficiency reasons. This command considers its arguments as comparison expressions or file tests and returns an exit status corresponding to the result of the comparison (0 for true, 1 for false).
- With version 2.02, Bash introduced the [[...]] *extended test command*, which performs comparisons in a manner more familiar to programmers from other languages. Note that **[[** is a keyword, not a command.

  Bash sees **`[[ $a -lt $b ]]`** as a single element, which returns an exit status.

  An **if/then** construct can contain nested comparisons and tests.

```
if [ condition-true ]
then
   command 1
   command 2
   ...
else
   command 3
   command 4
   ...
fi
```

Add a semicolon when 'if' and 'then' are on same line.

```
if [ -x "$filename" ]; then
```

Else if and elif

elif

`elif` is a contraction for else if. The effect is to nest an inner if/then construct within an outer one.

```
if [ condition1 ] ; then
   command1
   command2
   elif [ condition2 ] ; then
   command4
   command5
else
   default-command
fi
```

The **`if test condition-true`** construct is the exact equivalent of **`if [ condition-true ]`**. As it happens, the left bracket, **[** , is a token which invokes the **test** command. The closing right bracket, **]** , in an if/test should not therefore be strictly necessary, however newer versions of Bash require it.

5.9.1 File test operators

Returns true if...

-e	file exists
-f	file is a *`regular`* file (not a directory or device file)
-s	file is not zero size
-d	file is a directory
-b	file is a block device (floppy, cdrom, etc.)
-c	file is a character device (keyboard, modem, sound card, etc.)
-p	file is a pipe
-h	file is a symbolic link
-L	file is a symbolic link
-S	file is a socket
-t	file (descriptor) is associated with a terminal device

This test option may be used to check whether the `stdin` (**`[ -t 0 ]`**) or `stdout` (**`[ -t 1 ]`**) in a given script is a terminal.

-r file has read permission (*for the user running the test*)
-w file has write permission (*for the user running the test*)
-x file has execute permission (*for the user running the test*)
-g set-group-id (sgid) flag set on file or directory

If a directory has the *`sgid`* flag set, then a file created within that directory belongs to the group that owns the directory, not necessarily to the group of the user who created the file. This may be useful for a directory shared by a workgroup.

-u set-user-id (suid) flag set on file

A binary owned by *root* with *`set-user-id`* flag set runs with *root* privileges, even when an ordinary user invokes it. [1] This is useful for executables (such as **pppd** and **cdrecord**) that need to access system hardware. Lacking the *suid* flag, these binaries could not be invoked by a non-root user.

```
              -rwsr-xr-t     1 root         178236 Oct   2   2000
/usr/sbin/pppd
```

A file with the *`suid`* flag set shows an *s* in its permissions.

-k *`sticky bit`* set

Commonly known as the "sticky bit", the *save-text-mode* flag is a special type of file permission. If a file has this flag set, that file will be kept in cache memory, for quicker access. If set on a directory, it restricts write permission. Setting the sticky bit adds a *t* to the permissions on the file or directory listing.

```
drwxrwxrwt    7 root         1024 May 19 21:26 tmp/
```

If a user does not own a directory that has the sticky bit set, but has write permission in that directory, he can only delete files in it that he owns. This keeps users from inadvertently overwriting or deleting each other's files in a publicly accessible directory, such as `/tmp`.

-O	you are owner of file
-G	group-id of file same as yours
-N	file modified since it was last read
f1 -nt f2	file *f1* is newer than *f2*
f1 -ot f2	file *f1* is older than *f2*
f1 -ef f2	files *f1* and *f2* are hard links to the same file

"not" -- reverses the sense of the tests above (returns true if condition absent).

5.9.2 Comparison operators (binary)

integer comparison

-eq	is equal to	`if [ "$a" -eq "$b" ]`
-ne	is not equal to	`if [ "$a" -ne "$b" ]`
-gt	is greater than	`if ["$a" -gt "$b" ]`
-ge	is greater than or equal to	`if [ "$a" -ge "$b" ]`
-lt	is less than	`if [ "$a" -lt "$b" ]`
-le	is less than or equal to	`if [ "$a" -le "$b" ]`
<	is less than	`(("$a" < "$b"))`
<=	is less than or equal to	`(("$a" <= "$b"))`
>	is greater than	`(("$a" > "$b"))`
>=	is greater than or equal to	`(("$a" >= "$b"))`

string comparison

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

This operator uses pattern matching within a [[...]] construct.

<	is less than, in ASCII alphabetical order	`if [[ "$a" < "$b" ]]` `if [ "$a" \< "$b" ]`
	Note that the "<" needs to be escaped within a `[ ]` construct.	
>	is greater than, in ASCII alphabetical order	`if [[ "$a" > "$b" ]]`

```
if [ "$a" \> "$b" ]
```

Note that the ">" needs to be escaped within a **[ ]** construct.

-z string is "null", that is, has zero length

-n string is not "null".

Example : arithmetic and string comparisons

```
a=4
b=5
#  Here "a" and "b" can be treated either as integers or strings.
#    There  is  some  blurring  between  the  arithmetic  and  string
comparisons, since Bash variables are not strongly typed.

if [ "$a" -ne "$b" ]
then
  echo "$a is not equal to $b"
  echo "(arithmetic comparison)"
fi

echo

if [ "$a" != "$b" ]
then
  echo "$a is not equal to $b."
  echo "(string comparison)"
fi     # In this instance, both "-ne" and "!=" work.
```

5.9.3 Internal Variables

`$BASH` the path to the *Bash* binary itself, usually `/bin/bash`

`$EDITOR` the default editor invoked by a script, usually **vi** or **emacs**.

`$GROUPS` groups current user belongs to
This is a listing (array) of the group id numbers for current user, as recorded in `/etc/passwd`.

`$HOME` home directory of the user, usually `/home/username`

`$HOSTNAME` The hostname command assigns the system name at bootup in an init script. However, the `gethostname()` function sets the Bash internal variable `$HOSTNAME`.

`$OSTYPE` operating system type

```
bash$ echo $OSTYPE
linux-gnu
```

`$PATH` path to binaries, usually `/usr/bin/`, `/usr/local/bin`, etc.

When given a command, the shell automatically does a hash table search on the directories listed in the *path* for the executable. The path is stored in the environmental variable, `$PATH`, a list of directories, separated by colons. Normally, the system stores the `$PATH` definition in `/etc/profile` and/or `~/.bashrc`.

```
bash$ echo $PATH
```

```
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

`PATH=${PATH}:/opt/bin` appends the `/opt/bin` directory to the current path.

`$PWD` present working directory (directory you are in at the time)

Examples of showing the value of variables:

In the Bourne, BASH and Korn shell, to show the value of all your variables:

```
set
BASH=/usr/local/utils/bin/bash
BASH_VERSION=1.12.1
DELIVERY=John Smith Room 2004 JCMB
DISPLAY=frolix8.ucs:0
EDITOR=ue
FOLDERS=/home/eucs/erpl08/Mail
HISTFILE=/home/eucs/erpl08/.bash_history
HISTFILESIZE=50
HISTSIZE=100
HOME=/home/eucs/erpl08

...
```

This shows the value of an assorted range of variables.

5.10 Loops and Branches

Operations on code blocks are the key to structured, organized shell scripts. Looping and branching constructs provide the tools for accomplishing this.

5.10.1 Loops

A *loop* is a block of code that repeats a list of commands as long as the loop control condition is true.

5.10.1.1 for loop

for (in)

This is the basic looping construct. It differs significantly from its C counterpart.

for *arg* in [*list*]
do
 command...
done

The argument *list* may contain wild cards.

Example : Simple for loops

```
#!/bin/bash
# List the planets.

for planet in Mer Ven Earth Mars Jup Saturn Uranus Neptun Pluto
do
  echo $planet
done
```

Example : Operating on files with a for loop

```
for file in *.*
do
  ls -l "$file"  # Lists all files in $PWD (current directory).
done


for file in [jx]* ; do
  rm -f $file
# Removes only files beginning with "j" or "x" in $PWD.
  echo "Removed file \"$file\"".
done
```

Example : A C-like for loop

```
# Two ways to count up to 10.
for a in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$a "
done


# Now, let's do the same, using C-like syntax.
LIMIT=10
for ((a=1; a <= LIMIT ; a++))
  # Double parentheses, and "LIMIT" with no "$".
do
  echo -n "$a "
done                            # A construct borrowed from 'ksh93'.

#  Let's  use  the  C  "comma  operator"  to  increment  two  variables
simultaneously.
for ((a=1, b=1; a <= LIMIT ; a++, b++))
 # The comma chains together operations.
do
  echo -n "$a-$b "
done
```

5.10.1.2 while loop

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status).

while [*condition*]
do

command...
done

As is the case with for/in loops, placing the **do** on the same line as the condition test requires a semicolon.

while [*condition*] ; do

Example : Simple while loop

```
var0=0;LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
do
  echo -n "$var0 "        # -n suppresses newline.
  var0=`expr $var0 + 1`   # var0=$(($var0+1)) also works.
done
```

Example : Another while loop

```
while [ "$var1" != "end" ]     # while test "$var1" != "end"
do                             # also works.
  echo "Input variable #1 (end to exit) "
  read var1                    # Not 'read $var1' (why?).
  echo "variable #1 = $var1"   # Need quotes because of "#".
  echo
done
```

5.10.1.3 until loop

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (opposite of **while** loop).

until [*condition-is-true*]
do
command...
done

Note that an **until** loop tests for the terminating condition at the top of the loop, differing from a similar construct in some programming languages.

Example : until loop

```
until [ "$var1" = end ] # Tests condition here, at top of loop.
do
  echo "Input variable #1 "
  echo "(end to exit)"
  read var1
  echo "variable #1 = $var1"
done
```

5.11 Nested Loops

A nested loop is a loop within a loop, an inner loop within the body of an outer one. What happens is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes. Of course, a **break** within either the inner or outer loop may interrupt this process.

Example : Nested Loop

```
# Nested "for" loops.

outer=1             # Set outer loop counter.

# Beginning of outer loop.
for a in 1 2 3 4 5
do
  echo "Pass $outer in outer loop."
  echo "---------------------"
  inner=1           # Reset inner loop counter.

  # Beginning of inner loop.
  for b in 1 2 3 4 5
  do
    echo "Pass $inner in inner loop."
    let "inner+=1"  # Increment inner loop counter.
  done
  # End of inner loop.

  let "outer+=1"    # Increment outer loop counter.
  echo              # Space between output in pass of outer loop.
done
# End of outer loop.

exit 0
```

5.12 Branching construct

The **case** constructs is technically perhaps not a loop construct, since it does not iterate the execution of a code block. Like loops, however, it direct program flow according to conditions at the top or bottom of the block.

Controlling program flow in a code block

case (in) / esac

The **case** construct is the shell equivalent of **switch** in C/C++. It permits branching to one of a number of code blocks, depending on condition tests. It serves as a kind of shorthand for multiple if/then/else statements and is an appropriate tool for creating menus.

case "$*variable*" in

"$*condition1*")

command... ;;

"$*condition2*")
command... ;;

esac

- Quoting the variables is not mandatory, since word splitting does not take place.
- Each test line ends with a right parent).
- Each condition block ends with a *double* semicolon ;;.
- The entire **case** block terminates with an **esac** (*case* spelled backwards).

Example : Using case

```
echo;
echo "Hit a key, then hit return."
read Keypress

case "$Keypress" in
  [a-z]   ) echo "Lowercase letter";;
  [A-Z]   ) echo "Uppercase letter";;
  [0-9]   ) echo "Digit";;
  *       ) echo "Punctuation, whitespace, or other";;
esac  # Allows ranges of characters in [square brackets].
```

5.13 Examples of Bourne shell scripts

5.13.1 To read input to a command and process it in some way:

```
# usage: fsplit file1 file2
total=0; lost=0
while read next
do
total=`expr $total + 1`
case "$next" in
*[A-Za-z]*)  echo "$next" >> $1 ;;
*[0-9]*)     echo "$next" >> $2 ;;
*)           lost=`expr $lost + 1`
esac
done
echo "$total lines read, $lost thrown away"
```

The user types the command:

fsplit *file1 file2*

They then enter lines of text and issue an EOF instruction. The script then processes the lines as follows:

A line with at least one letter is appended to *file1*; any line with at least one digit and no letters is appended to *file2*. All other lines are thrown away.

5.13.2 To read commands from the terminal and process them:

```
# usage: process sub-directory
dir=`pwd`
for i in *
do
if test -d $dir/$i
then
  cd $dir/$i
  while echo "$i:"
  read x
  do
    eval $x
  done
  cd ..
fi
done
```

The user types the command:

process *sub-directory*

This script will read and process commands in the named `sub-directory`. The user is prompted to supply the name of the command to be **read** in. This command is executed using the the builtin **eval** function.

5.14 Summary

This chapter assumes no previous knowledge of scripting or programming, but progresses rapidly towards an intermediate/advanced level of instruction. It serves as a source of knowledge on shell scripting techniques.

Once you've logged in, you're working with a program called a shell. The shell interprets the commands you enter, runs the program you've asked for, and generally coordinates what happens between you and the UNIX operating system. Common shells include Bourne (sh), Korn (ksh), and C (csh) shells, as well as bash and tcsh.

For a beginner, the differences between most shells are slight. If you plan to do a lot of work with UNIX, though, ask your system administrator which shell your account uses; you should learn more about your shell and its set of special commands.

5.15 Check Your Progress

I Choose the correct answer

1) Shell is a _____.
   a) Interpreter
   b) Complier
   c) Assembler
   d) Linker
2) Shell scripts are n't useful when applications are_____
   a) complex
   b) portable
   c) secured
   d) all of the above
3) All LINUX systems have the following _____
   a) stdin
   b) stdout
   c) stderr
   d) all of the above
4) < redirects the _____
   a) stdin
   b) stdout
   c) stderr
   d) none of the above
5) chmod 777 scriptname gives _____ permission.
   a) Read/execute
   b) Read/write/execute
   c) Read/write
   d) Write/execute
6) The symbol used for command separator ____.
   a) ; b) ;; c) , d) .
7) Terminator in a case option ___ .
   a) ; b) , c) " d) ;;
8) _____runs job in background.
   a) Sleep 100
   b) Sleep 100 &
   c) & sleep 100
   d) sleep & 100
9) ____ is the name of the script.
   a) $1 b) $2 c) $0 d) none of the above
10) shift command _____.
   a) reassigns the positional parameters, in effect shifting them to the right one notch.
   b) reassigns the positional parameters, in effect shifting them to the left one notch.

c) reassigns the positional parameters, in effect shifting them to the both right & left one notch.
d) None of the above.

11) Pipelines are used to _____ commands.
a) reverse b) connect c) both a) & b) d) none of these

12) _____ is a synonym for test.
a)] b) [c) both a) & b) d) none of these

13) ["$a" –ne "$b"] is _____.
a) Arithmetic comparison
b) String comparison
c) Both a) & b)
d) Neither a) nor b)

14) Following are the Internal variables _____.
a) $HOME
b) $PATH
c) all of the above
d) none of the above

15) case test line ends with _____.
a)) b) (c))) d) none of the above.

II. Say True or False

1. exit status 0 results false. True/False
2. Bash an acronym for Bourne-Again Shell. True/False
3 The backquotes around the command illustrate the use of command replacement. True/False

III. Essay type questions

1) What is shell? Why shell programming is required.
2) When shell scripts are n't useful?
3) Breifly explain the access permissions related to differen types of users.
4) Elaborate all the looping control constructs.
5) Explain all the string related operators.
6) Write a simple shell script program for displaying current date,all logged-in users,process status using switch-case construct.
7) Write a shell script program to count the number of vowels,consonants and digits in a given text.
8) Write a shell script program to check whether a given string is palindrome or n't.
9) Explain all the test operators related to files.

IV. Further readings and other activities

1. Get more information using man or info or any help command for
ksh, sh, bash, tcsh, csh, ksh
EX: 1. $ man sh 2. $ apropos shell

Work on all of the example explained in the unit. Some exercises also are given there itself relating to the particular topic.

2. For further readings you can refer the following books
   1. Title: UNIX Shell Programming
      Author: Stephen G. Kochan and Patrick H. Wood; Howard Sams;

ISBN 0-672-48448-X; 1990. An excellent introduction to Bourne and Korn shell programming with lots of illustrative examples.

2. Title: Unix Shell Programming
   Author: Yashawant Kanitkar
   Publication: BPB

Reference e-mails: raomvp@yahoo.com roopasindhe@lycos.com
URL / Web Site: http://www.raomvp.bravepages.com

Good-Luck