

Chapter 20

Programming and Scripting

This chapter introduces the user to a comparison to various programming and scripting techniques. Examples are shown to highlight the differences between the various alternatives, showing where each might be stronger than another.

Concepts Learned in this Chapter

- Basic Scripting
- Various higher level programs that are available on Linux

Table of Contents

| | |
|--|----|
| Programming and Scripting..... | 1 |
| 20.1 Stream Editor | 3 |
| 20.1.1 Command Syntax..... | 3 |
| 20.1.2 Sed Examples..... | 6 |
| 20.2 Awk and Gawk | 7 |
| 20.3 Script Programming , | 9 |
| 20.3.1 Elementary Commands | 9 |
| 20.3.1.1 Echo Command | 9 |
| 20.3.1.2 Quotation Marks | 10 |
| 20.3.1.3 Variables | 11 |
| 20.3.1.4 Comparison Variables | 11 |
| 20.3.1.5 User Input Arguments | 12 |
| 20.3.1.6 Conditional Statements | 13 |
| 20.3.2 Computational Commands | 19 |
| 20.3.3 Running a Script | 20 |
| 20.3.2.1 Bash Script | 20 |
| 20.3.2.2 Stand-alone Execution | 20 |
| 20.4 Command Line Interpreter Programs | 21 |
| 20.4.1 Perl Programming Language | 21 |
| 20.4.2 Python Programming Language | 21 |
| 20.5 Compiled Programs | 22 |
| 20.5.1 C Programming Language | 22 |
| 20.5.2 C++ Programming Language | 22 |
| 20.5.3 Fortran Programming Language | 22 |
| 20.6 Web Based Languages | 22 |
| 20.6.1 PHP Programming Language | 22 |
| 20.6.2 Java Programming Language | 23 |
| 20.6.3 Ruby Programming Language | 23 |
| 20.7 Machine Language Programming | 23 |
| 20.8 Commands Used in this Chapter..... | 23 |
| 20.9 Chapter Review Questions..... | 23 |

To say the least, being able to program using various languages is very important to obtain the full benefit of an operating system. Unix and Linux normally contain many of the popular languages by default, assuming that they have been installed. The following is a short list, and some of the attributes of the languages.

It is not the intent to teach how to use the languages, but rather to learn some their attributes and that they are available.

20.1 Stream Editor

One of the strongest applications that is available for Unix and Linux is the **Stream Oriented Editor**, or **sed**. **Sed** is a non-interactive editor that takes its commands either from the command line or from a command file. Because **sed** is an editor, one might ask why is it considered in a programming chapter – that is because it has a set of commands that need to be learned. These are scripting directives. Only a very simple discussion of the power of **sed** will be discussed, as books have been written on the topic.

The objective of **sed** is to enact a set of commands against each line of data. A data file is read into a memory buffer, and then all of the **sed** commands are issued against each individual line of the data file. The objective of **sed** is to be able to repetitively issue a set of commands on each line of one or more files. After one line is processed, the next line processed. The output will be to the standard output (monitor) unless the output is redirected.

The two forms of the command are:¹

1. **sed “sed-commands” input-file > output-file**
2. **sed -f sed-script-file input-file > output-file**

In both of these cases, the output is saved to a new file.

20.1.1 Command Syntax

All versions of **sed** support Basic Regular Expressions (BREs).

A SED program consists of one or more SED commands, passed in by one or more of the **-e**, **-f**, **--expression**, and **--file** options, or the first non-option argument if zero of these options are used. This document will refer to "the" SED script; this will be understood to mean the in-order catenation of all of the **scripts** and **script-files** passed in.

Each SED command consists of an optional address or address range, followed by a one-character command name and any additional command-specific code.²

Addresses in a SED script can be in any of the following forms:

Addresses
`number`

¹ <http://sed.sourceforge.net/sedfaq.html>

² http://www.gnu.org/software/sed/manual/html_mono/sed.html

Specifying a line number will match only that line in the input. (Note that SED counts lines continuously across all input files.)

``first~step'`

This GNU extension matches every ***step***th line starting with line ***first***. In particular, lines will be selected when there exists a non-negative ***n*** such that the current line-number equals ***first*** + (***n*** * ***step***). Thus, to select the odd-numbered lines, one would use 1~2; to pick every third line starting with the second, 2~3 would be used; to pick every fifth line starting with the tenth, use 10~5; and 50~0 is just an obscure way of saying 50.

``$'`

This address matches the last line of the last file of input.

``/regexp'`

This will select any line which matches the regular expression ***regexp***. If ***regexp*** itself includes any / characters, each must be escaped by a backslash (\).

``\%regexp%'`

(The % may be replaced by any other single character.) This also matches the regular expression ***regexp***, but allows one to use a different delimiter than /. This is particularly useful if the ***regexp*** itself contains a lot of /s, since it avoids the tedious escaping of every /. If ***regexp*** itself includes any delimiter characters, each must be escaped by a backslash (\).

``/regexp/l'`

``\%regexp%/l'`

The l modifier to regular-expression matching is a GNU extension which causes the ***regexp*** to be matched in a case-insensitive manner.

If no addresses are given, then all lines are matched; if one address is given, then only lines matching that address are matched.

The use of ***sed*** requires knowledge of these basic commands:

Comment

``#'`

[No addresses allowed.] The # "command" begins a comment; the comment continues until the next newline. If you are concerned about portability, be aware that some implementations of SED (which are not POSIX.2 conformant) may only support a single one-line comment, and then only when the very first character of the script is a #. Warning: if the first two characters of the SED script are #n, then the -n (no-autoprint) option is forced. If you want to put a comment in the first line of your script and that comment begins with the letter `n' and you do not want this behavior, then be sure to either use a capital `N', or place at least one space before the `n'.

Substitute

``s/regexp/replacement/flags'`

(The / characters may be uniformly replaced by any other single character within any given s command.) The / character (or whatever other character is used in its stead) can appear in the **regexp** or **replacement** only if it is preceded by a \ character. Also newlines may appear in the **regexp** using the two character sequence \n. The s command attempts to match the pattern space against the supplied **regexp**. If the match is successful, then that portion of the pattern space which was matched is replaced with **replacement**. The **replacement** can contain \n (*n* being a number from 1 to 9, inclusive) references, which refer to the portion of the match which is contained between the *n*th \ (and its matching \). Also, the **replacement** can contain un-escaped & characters which will reference the whole matched portion of the pattern space. To include a literal \, &, or newline in the final replacement, be sure to precede the desired \, &, or newline in the **replacement** with a \. The s command can be followed with zero or more of the following **flags**:

Apply All Matches

'g'

Apply the replacement to **all** matches to the **regexp**, not just the first.

Print New Pattern

'p'

If the substitution was made, then print the new pattern space.

Replace Numbered Lines

'number'

Only replace the **number**th match of the **regexp**.

Write result to named file

'w file-name'

If the substitution was made, then write out the result to the named file.

Match Case-Insensitive

'I'

(This is a GNU extension.) Match **regexp** in a case-insensitive manner.

Terminate and Exit

'q'

[At most one address allowed.] Exit SED without processing any more commands or input. Note that the current pattern space is printed if auto-print is not disabled.

Delete Pattern Space

'd'

Delete the pattern space; immediately start next cycle.

Print Pattern Space

`p'

Print out the pattern space (to the standard output). This command is usually only used in conjunction with the -n command-line option. Note: some implementations of SED, such as this one, will double-print lines when auto-print is not disabled and the p command is given. Other implementations will only print the line once. Both ways conform with the POSIX.2 standard, and so neither way can be considered to be in error. Portable SED scripts should thus avoid relying on either behavior; either use the -n option and explicitly print what you want, or avoid use of the p command (and also the p flag to the s command).

Print Pattern Space and Replace

`n'

If auto-print is not disabled, print the pattern space, then, regardless, replace the pattern space with the next line of input. If there is no more input then SED exits without processing any more commands.

Multiple Command Input

`{ commands }'

A group of commands may be enclosed between { and } characters. (The } must appear in a zero-address command context.) This is particularly useful when you want a group of commands to be triggered by a single address (or address-range) match.

Append

`a' `text'

Append text to specified location.

Insert

`i' `text'

Immediately output the lines of text which follow this command (each but the last ending with a \, which will be removed from the output).

20.1.2 Sed Examples

The following are some examples of using sed

Double space a file

sed G file

Triple space a file

sed 'G;G' file

Under UNIX: convert DOS newlines (CR/LF) to Unix format

sed 's/\$// ' file # assumes that all lines end with CR/LF

sed 's/^M\$// ' file # in bash/tcsh, press Ctrl-V then Ctrl-M

Under DOS: convert Unix newlines (LF) to DOS format

sed 's/\$// ' file # method 1

```

sed -n p file                # method 2

# Delete leading whitespace (spaces/tabs) from front of each line
# (this aligns all text flush left). '^t' represents a true tab
# character. Under bash or tcsh, press Ctrl-V then Ctrl-L.
sed 's/^[^t]*//' file

# Delete trailing whitespace (spaces/tabs) from end of each line
sed 's/[^t]*$//' file        # see note on '^t', above

# Delete BOTH leading and trailing whitespace from each line
sed 's/^[^t]*//;s/[^t]*$//' file # see note on '^t', above

# Substitute "foo" with "bar" on each line
sed 's/foo/bar/' file        # replaces only 1st instance in a line
sed 's/foo/bar/4' file        # replaces only 4th instance in a line
sed 's/foo/bar/g' file        # replaces ALL instances within a line

# Substitute "foo" with "bar" ONLY for lines which contain "baz"
sed '/baz/s/foo/bar/g' file

# Delete all CONSECUTIVE blank lines from file except the first.
# This method also deletes all blank lines from top and end of file.
# (emulates "cat -s")
sed '/./,/^$/!d' file        # this allows 0 blanks at top, 1 at EOF
sed '/^$/N;/\n$/D' file      # this allows 1 blank at top, 0 at EOF

# Delete all leading blank lines at top of file (only).
sed '/./,$!d' file

# Delete all trailing blank lines at end of file (only).
sed -e :a -e '/^\n*$/{$d;N;};/\n$/ba' file

# If a line ends with a backslash, join the next line to it.
sed -e :a -e '/\$/N; s/\n//; ta' file

# If a line begins with an equal sign, append it to the previous
# line (and replace the "=" with a single space).
sed -e :a -e '$!N;s/\n=/ /;ta' -e 'P;D' file

```

20.2 Awk and Gawk

awk, or the newer version called **gawk**, is a simple programming paradigm – find a pattern in the input and then perform an action – often reduced complex or tedious data manipulations to few lines of code. The name **awk** comes from the

initials of its designers: Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan.³

There are several ways to run an awk program. If the program is short, it is easiest to include it in the command that runs awk, like this:

```
awk 'program' input-file1 input-file2 ...
```

When the program is long, it is usually more convenient to put it in a file and run it with a command like this:

```
awk -f program-file input-file1 input-file2 ...
```

A **comment** is some text that is included in a program for the sake of human readers; it is not really an executable part of the program. Comments can explain what the program does and how it works. Nearly all programming languages have provisions for comments, as programs are typically hard to understand without them.

In the **awk** language, a comment starts with the sharp or pound sign character (#) and continues to the end of the line. The # does not have to be the first character on the line. The **awk** language ignores the rest of a line following a sharp sign. For example:

```
# This program prints a nice friendly message. It helps  
# keep novice users from being afraid of the computer.  
BEGIN { print "Don't Panic!" }
```

For short to medium length awk programs, it is most convenient to enter the program on the awk command line. This is best done by enclosing the entire program in single quotes. This is true whether you are entering the program interactively at the shell prompt, or writing it as part of a larger shell script:

```
awk 'program text' input-file1 input-file2 ...
```

Once you are working with the shell, it is helpful to have a basic knowledge of shell quoting rules. The following rules apply only to POSIX-compliant, Bourne-style shells (such as bash, the GNU Bourne-Again Shell). If you use csh, you're on your own.

- Quoted items can be concatenated with non-quoted items as well as with other quoted items. The shell turns everything into one argument for the command.
- Preceding any single character with a backslash (\) quotes that character. The shell removes the backslash and passes the quoted character on to the command.
- Single quotes protect everything between the opening and closing quotes. The shell does no interpretation of the quoted text, passing it on verbatim to the command. It is *impossible* to embed a single quote inside single-quoted text. Refer back to [Comments](#), for an example of what happens if you try.
- Double quotes protect most things between the opening and closing quotes. The shell does at least variable and command

³ http://www.gnu.org/software/gawk/manual/html_mono/gawk.html

substitution on the quoted text. Different shells may do additional kinds of processing on double-quoted text.

Since certain characters within double-quoted text are processed by the shell, they must be *escaped* within the text. Of note are the characters \$, `, \, and ", all of which must be preceded by a backslash within double-quoted text if they are to be passed on literally to the program. (The leading backslash is stripped first.) The following example illustrates this concept:

```
$ awk "BEGIN { print \"Don't Panic!\" }"
outputs:
-| Don't Panic!
```

Note that the single quote is not special within double quotes.

- Null strings are removed when they occur as part of a non-null command-line argument, while explicit non-null objects are kept. For example, to specify that the field separator FS should be set to the null string, use:
`awk -F "" 'program' files # correct`

20.3 Script Programming ^{4,5}

Each of the shells within Unix and Linux maintain a basic programming capability. This level of programming, which may be performed directly from the command line, are called **scripts**. They are typically very simple, but are capable of immense power. Since the default shell for Linux is **bash**, we will review some of the basic powers of the language. Although scripting is a very powerful language, it has many attributes, of which only a small fraction of which are able to be discussed here.

Both Unix and Linux rely very heavily on a multitude of scripts during the boot process, which perform a number of branches to enable various processes and services. The user may also expand upon the basic process by creating scripts to do repetitive tasks.

20.3.1 Elementary Commands

We will review a short list of the most common commands available, and some of the special coding required.

20.3.1.1 Echo Command

Before one gets into actual programming commands, a review of the one command that outputs to the stdout (monitor) is required. This is important because it is the primary means to display information on the screen. The syntax of the command is:

echo (option) “text to be displayed”

Options include:

⁴ <http://www.gnu.org/software/bash/manual/bashref.html>

⁵ **Hello Linux**, by Clyde Boom, Lancom Technologies, ISBN 1-896814-22-0

- n Do not output the training newline
 - the newline is equivalent to a carriage return (CR) and linefeed (LF), where the cursor is moved to the beginning of the next line.
- e Enable interpretation of the backslash-escaped characters (explained below)
- E Disable interpretation of the backslash-escaped characters in strings (in quotes)

Escape characters include (when the -E option is not specified):

- \NNN The character whose ASCII code is NNN (specified in octal)
- \\ Keep the backslash character in the text
- \a Alert – ring the bell
- \b Backspace the cursor one character
- \c Suppress trailing newline at the end of the input
- \f Issue a form feed to start a new page
- \n Start a newline (Carriage Return + Line Feed)
- \r Issue a Carriage Return only without a Line Feed
- \t Issue a horizontal tab
- \v Issue a vertical tab

20.3.1.2 Quotation Marks

In order to insure the proper stdout (monitor) display in conjunction with the **echo** command, we need to use quotation marks. There are three different types quotation marks available.

- **Single Quotes – ‘ ’** Used around text that is to be displayed as it is written. For example:

```
echo 'You are user $USER' returns
You are user $USER
```

Here the variable \$USER is not replaced.

- **Backtick – ` `** Used around commands that are to be processed. For example:

```
echo "Today's date is `date`" returns
Today's date is Mon Feb 16 10:36 CST 2004
```

In this case, the command “date” is issued and the date and time is returned.

- **Double Quotes – “ ”** Used around text where variables are to be processed. For example:

```
echo "You are user $USER and today is `date`"
You are user dennis and today is Mon Feb 16 10:40 CST
2004
```

In this example, the variable \$USER is displayed and the command date is enacted.

20.3.1.3 Variables

Variables may be assigned within a script, or the Environmental variables may be used. Environmental variables are preset and may be viewed by using the **env** command.

To set a user variable, the user specifies that the specified variable name is equal to what they wish. The value may be either a word or a numeric value.

```
mday = Monday
tday = Tuesday
wday = Wednesday
```

To display the variable, the variable name is preceded with the '\$', such as:

```
echo Today is $mday
Today is Monday
echo Tomorrow is $tday
Tomorrow is Tuesday
```

One must be cautious when using double quotes because extra spaces or tab characters included in a variable are truncated to only one space. For example if the variable is:

```
wdays1 = Monday Tuesday Wednesday Thursday Friday
wdays2 = "Monday Tuesday Wednesday Thursday
Friday"
echo wdays1
Monday Tuesday Wednesday Thursday Friday
echo wdays2
Monday Tuesday Wednesday Thursday Friday
```

20.3.1.4 Comparison Variables

Variables may be tested to determine if a condition exists.

➤ Directory and File Comparison

| | | |
|-----------|------|--|
| -a | item | True if file exists |
| -b | item | True if the file exists and it is of type block |
| -c | item | True if the file exists and it is of type character |
| -d | item | True if the item exists and is a directory |
| -e | item | True if the item (directory or file) exists |
| -f | item | True if the item exists and is a regular file |
| -g | item | True if the item exists and its set-group-id is set |
| -k | item | True if the item exists and its sticky bit is set |
| -h | item | True if the item exists and it is a symbolic link |
| -r | item | True if the item exists and it is readable |
| -s | item | True if the item exists and the length is greater than zero |
| -u | item | True if the item exists and the set-user-id bit is set |
| -w | item | True if the item exists and the file is writeable |
| -x | item | True if the item exists and is executable |
| -G | item | True if the item exists and is owned by the effective group that the user is a member of |

- O item True if the item exists and is owned by the current logged in user
- Numerical Comparison
 - eq True if first number is equal to second
 - ne True if first number is not equal to second
 - gt True if first number is greater than second
 - ge True if first number is greater than or equal to second
 - lt True if first number is less than second
 - le True if first number is less than or equal to second
- String Comparison
 - \$string1 = \$string2** True if string1 is equal to string2
 - \$string1 != \$string2** True if string1 is not equal to string2
 - n \$string** True if string is of non-zero length
 - z \$string** True if string is of zero length
- Logical Comparison
 - ! expression** True if the expression is FALSE
False if the expression is TRUE
 - expression1 -a expression2** True if expression1 **AND** expression2 are both TRUE, **-a** represents the logical AND function
 - expression1 -o expression2** True if expression1 **OR** expression2 is TRUE, **-o** represents the logical OR function

If a test for a condition is to be performed prior to performing a task, then the following format may be utilized:

[-f path1/file1] && . /path2/command

This form has the following meaning

[-f path1/file1] Test to see if file1 in path exists
&& Conditional AND, perform the following if the previous condition is true
“.” (space) Run the following command specified by
/path2/command Command or script to be executed

20.3.1.5 User Input Arguments

Quite often, a command is followed by arguments that are specified by the user. Special characters are assigned to specify the command arguments. As an example, assume that a script by the name of display has been written to echo the arguments that follow it, such as:

Script display

echo “\$0” Displays the original command or script name
echo “\$1” Displays the first argument
echo “\$2” Displays the second argument
echo “\$3” Displays the third argument
echo “\$@” Displays all of the arguments
echo “\$#” Displays the number of arguments

Using the script:

```
Script display
echo "$0"
echo "$1"
echo "$2"
echo "$3"
echo "$@"
echo "$#"
```

Thus if we issue the command:

```
$display red blue green
```

the output is:

```
display
red
blue
green
red blue green
3
```

20.3.1.6 Conditional Statements

Several conditions commands are available to test a value and then execute only if true.

20.3.1.6.1 For Loop **for ... in ... do**

The for loop will repetitively cycle through a list of arguments. The syntax of the conditional statement is:

```
for Loop-Index in Argument-List
do
  statements to be performed
done
```

This performs the following actions:

1. Loop-Index is a range or list of items to be sorted through
2. Argument-List is a list of items that are to be tested
3. Statements is a list of commands that are to be performed

As an example:

```
Script colorlist
for colors in red blue green
do
  echo "$colors"
done
```

Outputs the following:

```
$ colorlist
red
blue
green
```

20.3.1.6.2 If Condition **if ... then**

The if condition tests a conditional clause, and if true then proceeds with the statements. If the condition is false, the statements are skipped. The syntax of the conditional statement is:

```
if condition-statement
then
    statements to be performed
fi
```

This performs the following actions:

1. Tests the condition-statement
2. If condition-statement is true, then
3. Perform 'statements to be performed'
4. If condition statement is false, then exit

As an example:

```
Script dirlab
if [ -x /lab ]
then
    echo The directory /lab exists
fi
```

Outputs the following:

```
$ dirlab
The directory /lab exists
```

20.3.1.6.3 If Else Condition **if ... then ... else**

The if – else condition tests the conditional clause. If true the first set of statements is completed, but if false, the second set of statements is completed. The syntax of the conditional statement is:

```
if condition
then
    first statements to be performed
else
    second statements to be performed
fi
```

This performs the following actions:

1. Tests the condition-statement
2. If condition-statement is true, then
3. Perform 'first statements to be performed'

4. If condition statement is false, then divert to the else section
5. Perform 'second statements to be performed'
6. Exit

As an example:

```
Script truestate
string1 = "today"
string2 = "tomorrow"
if [ $string1 = $string2 ]
then
    echo "$string1 and $string2 are the same"
else
    echo "$string1 and $string2 are not the same"
fi
```

Outputs the following:

```
$ truestate
today and tomorrow are not the same
```

20.3.1.6.4 If – Elself Condition if ... then ... elif

The if – elif condition tests the first condition and if true, performs the first statements. If it is false, the embedded second if condition is enacted, testing the second condition, which if true then the second set of statements is enacted. The syntax of the conditional statement is:

```
if condition1
then
    first statements to be performed
elif condition2
then
    second statements to be performed
fi
```

This performs the following actions:

1. Tests the condition1 statement
2. If condition1 is true, then
3. Perform 'first statements to be performed'
4. Exits if condition
5. If condition1 is false, then drop to secondary conditional test
6. If condition2 is true, then
7. Perform 'second statements to be performed'
8. Exits if condition
9. If condition2 is false, then exit if condition

This is a "nested" conditional test. If the condition is true, run the first statements, if the condition is false, then run the second statements.

As an example:

```

Script truestate
string1 = "today"
string2 = "tomorrow"
string3 = "today"
if [ $string1 = $string2 ]
then
    echo "$string1 and $string2 are the same"
    elif [ $string1 = $string3 ]
    then
        echo "$string1 and $string3 are the same"
fi

```

Outputs the following:

```

$ truestate
today and today are the same

```

This performs the following actions:

1. Tests the condition1 statement
2. If condition1 is true, then
3. Perform 'first statements to be performed'
4. Exits if condition
5. If condition1 is false, then test if second condition2 is true,
6. If condition 2 is true, then
7. Perform 'second statements to be performed'
8. Exits if condition
9. If condition2 is false, then exit if condition

Nested conditions may be set to any depth that is required – but may be confusing if more than three. Each nesting may also be combined with the else condition, although the else applies to the preceding conditional test.

20.3.1.6.5 While Loop **while . . . do**

The while loop process a set of statements while the specified condition is true. The syntax of the conditional statement is:

```

while condition
do
    statements to be performed
done

```

This performs the following actions:

1. The condition is tested, if true then
2. Perform the 'statements to be performed'
- If condition is false, then exit

As an example:

```

Script weekday
today = mon

```



```

while [ $today = mon ]
do
    echo Today is Monday
done

```

Outputs the following:

```

$ weekday
Today is Monday

```

This is also an excellent means to test a numeric value to and continually evaluate the condition until the evaluation is no longer true.

20.3.1.6.6 Until Loop **until . . . do**

The until loop processes a set of statements until the specified condition is true. The syntax of the conditional statement is:

```

until condition
do
    statements to be performed
done

```

This performs the following actions:

1. The condition is tested, if true then
2. Perform the 'statements to be performed'
3. If condition is false, then exit

This is the inverse of the while condition. The action statements are to be performed until the condition is true.

As an example:

```

Script j4
j = 2
until [ $j -eq 4 ]
do
    echo The value of j is $j
    j = (($j + 1))
done

```

Note that for the line “j = ((\$j + 1))”, this is an arithmetic expansion. In this case, the variable is assigned the value of what the existing variable j is, plus 1.

Outputs the following:

```

$ j4
The value of j is 2
The value of j is 3

```

20.3.1.6.7 Case Selection **case . . . in . . . pattern**

The case selection will select a value from the specified pattern and issue the statements relative to that set of commands. The syntax of the conditional statement is:

```

case test-string in
  pattern1
    first statements to be performed
    ;;
  pattern2
    second statements to be performed
    ;;
  pattern3
    third statements to be performed
    ;;
  . . .
esac

```

This performs the following case tests:

1. Variable test-string is tested to see if it is equal to pattern1
2. If true, 'first statements to be performed' are completed
3. Case statement is exited
4. If false, test-string is tested to see if it is equal to pattern2
5. If true, 'second statements to be performed' are completed
6. Case statement is exited
7. If false, test-string is tested to see if it is equal to pattern3
8. If true, 'third statements to be performed are completed
9. Case statement is exited
10. If false, case statement is exited

As an example:

```

Script comncase
clear
echo -e "      \n USER MENU\n"
echo "  a. List contents of the /lab directory"
echo "  b. Show attributes of the files in the /lab directory"
echo -e "  c. Display the full attributes of the /lab directory\n"
echo -e "Type in a, b, or c : \c"
read inkey
case "$inkey" in
  a)
    ls /lab
    ;;
  b)
    ls -l /lab/*
    ;;
  c)
    stat /lab
    ;;
  *)
    echo "Your entry of \" $inkey \" is not valid"
    ;;
esac

```

esac

Note that in this example that we have a new command, “read”. This is used to accept input from the keyboard that the user types in.

Outputs the following:

\$ comncase

USER MENU

- a. List contents of the /lab directory***
- b. Show attributes of the files in the /lab directory***
- c. Display the full attributes of the /lab directory***

Type in a, b, or c : d
Your entry of “ d ”is not valid

A non-quoted backslash ‘\’ is the Bash escape character. It preserves the literal value of the next character that follows, with the exception of **newline**. If a **\newline** pair appears, and the backslash itself is not quoted, the **\newline** is treated as a line continuation (that is, it is removed from the input stream and effectively ignored).

Backslash escape sequences, if present, are decoded as follows:

| | |
|-------------|--|
| \a | alert (bell) |
| \b | backspace |
| \e | an escape character (not ANSI C) |
| \f | form feed |
| \n | newline |
| \c | wait for input |
| \r | carriage return |
| \t | horizontal tab |
| \v | vertical tab |
| \\ | backslash |
| \' | single quote |
| \" | double quote |
| \nnn | the eight-bit character whose value is the octal value <i>nnn</i> (one to three digits) |
| \xHH | the eight-bit character whose value is the hexadecimal value <i>HH</i> (two hex digits) |
| \cx | a control-x character |

20.3.2 Computational Commands

Bash is also capable of supporting many computational functions. Some of the computational functions include:

| | |
|--------------|--|
| var++ | Increment variable by one |
| var-- | Decrement variable by one |
| + | Add two values or variables |
| - | Subtract the second value or variable from the first |

| | |
|-------------------|---|
| ** | Exponentially raise the first value or variable to the power of the second |
| * | Multiply the first value or variable by the second |
| / | Divide the first value or variable by the second |
| % | Divide the first value or variable by the second, retain only the remainder |
| == | Test the first value or variable to see if it is equal to the second |
| != | Test the first value or variable to see if it is not equal to the second |
| & | Bitwise, AND the first value or variable with the second |
| | Bitwise, OR the first value or variable with the second |
| ^ | Bitwise, Exclusive OR the first value or variable with the second |
| && | Logical AND the first value or variable with the second |
| | Logical OR the first value or variable with the second |
| = | Assign the second value or variable to the first |

These computational functions may be applied in the same manner as any other mathematical process.

20.3.3 Running a Script

After one has written their script, it needs to be run. There are several ways that it may be executed.

20.3.2.1 Bash Script

Since the script has been written, the easiest way to test and run it is by using the bash command:

```
bash scriptfile.sh
```

From this command, the script will be executed.

Although not necessary, it is convention to terminate a shell script with a “.sh”. From this the user may easily scan a list of files and recognize those that are shell scripts.

20.3.2.2 Stand-alone Execution

Naturally it is not desired to always have to issue the bash command prior to executing a script, especially if it is being called from another script or program.

The first requirement is to specify what the script is to be executed by. To do this the very first line of the script contains the command sequence:

```
#!/bin/bash
```

Previously you have been told that the “#” meant that the line was a comment, but in the case of a script for the first line only, this is not true – because the “!” follows. This character sequence is often called the “Shabang”. (Recall that the “!” is commonly called the “bang”.) Since all shell interpreters are located in the /bin directory, the “/bin/bash” specifies that the script will be executed using the bash command that is located in the /bin directory.

In order to make the script executable, the permissions must be modified. Recall that each entity, owner, group and world, each have permissions of read (r), write (w), and execute (x). The execute permission must be changed to an x value rather than a dash (-). This is done by issuing the command:

```
chmod a+x scriptfile.sh
```

In one command the execute permission has been modified to allow all (owner, group and world) to execute the command.

To verify this, issue the command:

```
ls -l scriptfile.sh
```

and you will observe that the execute permission has been modified for all users.

To issue the script from the command line, we need to specify the path to the script and the script name. For example:

```
$ /lab/scriptname.sh
```

would execute the script file “scriptname.sh” in the /lab directory.

If we are in the /lab directory, then we do not need to specify the full path, but we must tell the system where to start its search for the script file. In this case we issue the command:

```
$ ./scriptname.sh
```

where the dot “.” represents our present location and the slash specifies this directory. Note that there is no space between the dot and the slash.

20.4 Command Line Interpreter Programs

Several programs are available to the user to that allow simple programming, yet provide very powerful tools. Two such programs are **perl** and **python**. Both have a common base to the bash scripting, allowing for slightly different syntax of commands, but with much more power.

20.4.1 Perl Programming Language

The Perl language has a strong similar structure to the bash script – but it does have its differences. Knowledge of the C Language and of bash lends to the learning curve, but they are not necessary. In this discussion, we will focus on the minor differences and improvements that Perl provides over bash.

20.4.2 Python Programming Language

In a similar way, the Python language also has a lot of similarities to bash and the C language. But it too has its own set of differences that make it unique. Here again we will focus on the benefits of using the Python language.

20.5 Compiled Programs

Unix and Linux do not lack in advanced programming languages. In fact both Unix and Linux are based on the **C** Language. In addition, **C++** and **Fortran** (for those that remember that ancient language) are also available.

20.5.1 C Programming Language

The C Language was originally written back in the 1960's as a new approach to programming. It brought a major improvement to programming style, with an emphasis on a strong structured format. Additionally it provides features that allow the user to access features of the operating system that were previously unavailable. Obviously, the discussion what features are available is way beyond the scope of this text. Many books have been written and college courses dedicated to the topic. For the user that has the need to write C language programs, it is here.

A few words need to be noted. The C language was used to write the Unix Operating System, and if you wish to recompile the Kernel, you need to use the Unix / Linux version. This is because the Microsoft version does not comply with the full standards that the Unix / Linux version does.

20.5.2 C++ Programming Language

The C++ Language is an enhancement of the C language. The objective of using C++ is to allow object oriented programming, which provides the ability to include the system environment within the language. This provides additional features that may be used to enhance the writing of code. Again, discussion of this language is far beyond the scope of this text. It too has many college courses dedicated to this topic.

20.5.3 Fortran Programming Language

The Fortran Programming Language might be considered a very outdated language, but there are many programs that have been written that are still operational. It does not have all of the advanced features of more modern languages, and thus has been relegated to being a language that is rarely taught in today's education, and few, if any books are still published about it. If one is to program in Fortran, then some dedicated research will be required to find material on how to use the language may be required.

20.6 Web Based Languages

For the days of the Internet, use of the World Wide Web to find information is at the forefront of our life. Thus several different languages have been developed to take advantage of the web page environment.

20.6.1 PHP Programming Language

20.6.2 Java Programming Language**20.6.3 Ruby Programming Language****20.7 Machine Language Programming**

The base for all programming languages is Assembler – one step above machine language. Absolute total power for processing speed and compactness is available in assembler, at a cost of the programmer having to memorize a large number of acronyms for data manipulation. The knowledge of assembler is well beyond the scope of this text, thus will not be reviewed. Least it be said, in performance, assembler is the fastest of all languages.

The application for running assembler is:

as filename

20.8 Commands Used in this Chapter

| | |
|---------|--|
| AS | Programming Language |
| awk | Editing Programming Language |
| bash | Command shell and shell programming language |
| C | Programming Language |
| C++ | Programming Language |
| chmod | Modifies the permissions of a file |
| echo | Echos attached string to standard output |
| Fortran | Programming Language |
| gawk | GNU Editing Programming Language |
| Java | Programming Language |
| ls -l | List file's attributes |
| Perl | Programming Language |
| PHP | Programming Language |
| Python | Programming Language |
| Ruby | Programming Language |
| sed | Stream Editor |

20.9 Chapter Review Questions

Chapter Index

| | | | |
|-----------------------------------|----|------------------------------|----|
| A | | C++ | 22 |
| Assembler Language | 23 | Fortran | 22 |
| Awk | 7 | Java | 23 |
| B | | Perl | 21 |
| Bash Script | 20 | PHP | 22 |
| C | | Python | 21 |
| C Language | 22 | Ruby | 23 |
| C++ Language | 22 | M | |
| Case Selection Condition | 17 | Machine Language Programming | 23 |
| Command Line Interpreter Programs | 21 | P | |
| Compiled Programs | 22 | Perl Language | 21 |
| Computational Commands | 19 | PHP Language | 22 |
| Conditional Statement | | Python Language | 21 |
| Case Selection | 17 | R | |
| Computational Commands | 19 | Ruby Language | 23 |
| For Loop | 13 | S | |
| If - Elself | 15 | Script | 20 |
| If Condition | 14 | Script Execution | 20 |
| If Else | 14 | Script Programming | 9 |
| Until Loop | 17 | Scripts | |
| While Loop | 16 | Comparison Variables | 11 |
| Conditional Statements | 13 | Conditional Statements | 13 |
| F | | Echo Command | 9 |
| For Loop | 13 | Elementary Commands | 9 |
| Fortran Language | 22 | Programming | 9 |
| G | | Quotation Marks | 10 |
| Gawk | 7 | User Input | 12 |
| I | | Variables | 11 |
| If - Elself Condition | 15 | sed | 3 |
| If Condition | 14 | Shabang | 20 |
| If Else Condition | 14 | Stream Editor | 3 |
| J | | U | |
| Java Language | 23 | Until Loop Condition | 17 |
| L | | W | |
| Language | | Web Based Languages | 22 |
| Assembler | 23 | While Loop Condition | 16 |
| C | 22 | # | |
| | | #! | 20 |