just as if they had been typed on separate lines. If it is necessary to continue a command onto the next line, end the line with a backslash (\), press *Enter* and continue typing on the next line.

In a script anything after# on a line is treated as a comment, *i.e.,* it is ignored.

To background a command simply add an ampersand (&) on the end.
Commands may be piped using a vertical bar | to separate them. As an alternative to piping the output may be **redirected** to write to a file using > filename or appended to the file using >> filename. This only redirects the **standard output** stream; redirection of error messages (**standard error** stream) requires a different syntax which is shell-dependent. The **standard input** stream may be redirected to read from a file using <filename.

1.4.5 Getting Help

UNIX is an operating system, with hundreds of commands that can be combined to execute thousands of possible actions. UNIX provides complete information about each and every command which is stored in the UNIX *man* pages (*man* stands for manual). We can go through them by giving the *man* command at the prompt as shown below:

$ man cp

$ man ls

$ man grep

1.5 UNIX COMMANDS

With many competing standards (UNIX 98, UNIX95, POSIX.2, SVID3, 4.3BSD, etc.) and most users having to deal with multiple systems, it's crucial to know which commands are important enough to be used on nearly every version of UNIX. The following is a list of **commonly used commands** which are organised under different categories for understanding and ease of use. Keys proceeded by a ^ character are CONTROL key combinations.

Terminal Control Characters

^h	backspace erase previously typed character
^u	erase entire line of input so far typed
^d	end-of-input for programs reading from terminal
^s	stop printing on terminal
^q	continue printing on terminal
^z	currently running job; restart with bg or fg
DEL, ^c	kill currently running program and allow clean-up before exiting
^\	emergency kill of currently running program with no chance of cleanup

Login and Authentication

login	access computer; start interactive session
logout	disconnect terminal session
passwd	change local login password; you MUST set a non-trivial password

Information

date	show date and time
history	list of previously executed commands
pine	send or receive mail messages
msgs	display system messages
man	show on-line documentation by program name

info	on-line documentation for GNU programs
w, who	who is on the system and what are they doing
who am i	who is logged onto this terminal
top	show system status and top CPU-using processes
uptime	show one line summary of system status
finger	find out info about a user@system

File Management

cat	combine files
cp	copy files
ls	list files in a directory and their attributes
mv	change file name or directory location
rm	remove files
ln	create another link (name) to a file
chmod	set file permissions
des	encrypt a data file with a private key
find	find files that match specified criteria

Display Contents of Files

cat	copy file to display device
vi	screen editor for modifying text files
more	show text file on display terminal with paging control
head	show first few lines of a file(s)
tail	show last few lines of a file; or reverse line order
grep	display lines that match a pattern
lpr	send file to line printer
pr	format file with page headers, multiple columns etc.
diff	compare two files and show differences
cmp	compare two binary files and report if different
od	display binary file as equivalent octal/hex codes
file	examine file(s) and tell you whether text, data, etc.
wc	count characters, words, and lines in a file

Directories

cd	change to new directory
mkdir	create new directory
rmdir	remove empty directory (remove files first)
mv	change name of directory
pwd	show current directory

Devices

df	summarize free space on disk device
du	show disk space used by files or directories

Special Character Handling for C-shell

*	match any characters in a file name
~user	shorthand for home directory of "user"
$name	substitute value of variable "name"
\	turn off special meaning of character that follows
‘	In pairs, quote string w/ special chars, except !
“	In pairs, quote string w/ special chars, except !, $
`	In pairs, substitute output from enclosed command

Controlling Program Execution for C-shell

&	run job in background
DEL, ^c	kill job in foreground
^z	suspend job in foreground
fg	restart suspended job in foreground
bg	run suspended job in background
;	delimit commands on same line
( )	group commands on same line

!	re-run earlier command from history list
ps	print process status
kill	kill background job or previous process
nice	run program at lower priority
at	run program at a later time
crontab	run program at specified intervals
limit	see or set resource limits for programs
alias	create alias name for program (in .login)
sh, csh	execute command file

Controlling Program Input/Output for C-shell

|	pipe output to input
>	redirect output to a storage file
<	redirect input from a storage file
>>	append redirected output to storage file
tee	copy input to both file and next program in pipe
script	make file record of all terminal activity

E-mail and communication

pine	process mail with full-screen menu interface or read USENET news groups
msgs	read system bulletin board messages
mail	send e-mail; can be run by other programs to send existing files via e-mail
uuencode	uudecode encode/decode a binary file for transmission via mail
finger	translate real name to account name for e-mail
talk	interactive communication in real-time
rn	read USENET news groups

Editors and Formatting Utilities

sed	stream text editor
vi	screen editor
emacs	GNU emacs editor for character terminals
xemacs	GNU emacs editor for X-Windows terminals
pico	very simple editor, same as used in "pine"
fmt	fill and break lines to make all same length
fold	break long lines to specified length

Printing

lpr	send file to print queue
lpq	examine status of files in print queue
lprm	remove a file from print queue
enscript	convert text files to PostScript format for printing

Interpreted Languages and Data Manipulation Utilities

sed	stream text editor
perl	Practical Extraction and Report Language
awk	pattern scanning and processing language; 1985 vers.
sort	sort or merge lines in a file(s) by specified fields
tr	translate characters
cut	cut out columns from a file
paste	paste columns into a file
dd	copy data between devices; reblock; convert EBCDIC

Networking/Communications

telnet	remote network login to other computer
ftp	network file transfer program
rlogin	remote login to “trusted” computer
rsh	execute single command on remote “trusted” computer
rcp	remote file copy to/from “trusted” computer
host	find IP address for given host name, or vice versa

Switching between certain shells of the same syntax is a lot easier than switching between shells of a different syntax. So if you haven't much time a simple upgrade (e.g., csh to tcsh) may be a good idea.

Can I afford any minor bugs?

Like most software all shells have some bugs in them (especially csh), you can afford the problems that may occur because of them.

Do you need to be able to use more than one shell?

If you use more than one system then you may need to know more than one shell at the same time. How different are these two shells and can you manage the differences between them?

1.7.3 Shell Command files

It is possible to repeat a set of commands many times; as if the whole set were just a single command. UNIX allows you to put all your commands in a file in proper order, and then execute them one by one. When you type the file name, shell reads its contents and starts executing the commands in it, one by one. The file is called a shell file and the sequence of commands in it may be called a **shell program** or a **shell procedure**. Let us create a shell file called ***shellp*** (we can give any relevant file name) which contains UNIX commands:

who am i and ***date***

$ cat shellp
who am i
date

To execute this shell program ***shellp,*** first of all we to change the mode of the file using the ***chmod*** command, so that it will be executable.

$ chmod 777 shellp

Then execute the shell program by giving the command as follows:

$ shellp

Let us see how to write the Bourne shell scripts in the following section.

1.8 BOURNE SHELL PROGRAMMING

The focus of this section is to get you to understand and run some Bourne shell scripts. There are example scripts for you to run. Historically, people have been biased towards the Bourne shell over the C shell because in the early days the C shell was buggy. These problems are fixed in many C shell implementations, however many still prefer the Bourne shell. You can write shell programs by creating scripts containing a series of shell commands. The first line of the script should start with **#!** which indicates to the kernel that the script is directly executable. You immediately follow this with the name of the shell, or program (spaces are allowed), to execute, using the full path name. Generally, you can count on having up to 32 characters, possibly more on some systems, and can include one option. So to set up a Bourne shell script the first line would be:

#! /bin/sh
or for the C shell:
#! /bin/csh

You also need to specify that the script is executable by setting the proper bits on the file with **chmod**, e.g.:

% chmod +x shell_script

To introduce ***comments*** within the scripts use **#** (it indicates a comment from that point until the end of the line).

Shell scripting involves chaining several UNIX commands together to accomplish a task. For example, you might run the 'date' command and then use today's date as part of a file name. Let us see how to do this below:

To try the commands below start up a Bourne shell:

Example:

```
/bin/sh
#A variable stores a string (try running these commands in a Bourne shell)
name= "vvs"
echo $name
```

The quotes are required in the example above because the string contains a special character (the space)

A variable may store a number as:
num =137

The shell stores this as a string even though it appears to be a number. A few UNIX utilities will convert this string into a number to perform arithmetic:

expr $num + 3

Try defining num as '7m8' and try the *expr* command again
What happens when num is not a valid number?
Now you may exit the Bourne shell with
exit

I/O Redirection

The *wc* command counts the number of lines, words, and characters in a file
wc /etc/passwd
wc -l /etc/passwd

You can save the output of ***wc*** (or any other command) with output redirection
wc /etc/passwd > wc.file

You can specify the input with input redirection
wc < /etc/passwd

Many UNIX commands allow you to specify the input file by name or by input redirection
sort /etc/passwd
sort < /etc/passwd

You can also append lines to the end of an existing file with output redirection
wc -l /etc/passwd >> wc.file

Splitting a file

It may happen that the file you are handling is huge and takes too much time to edit. In such a case you might feel that the file should be split into smaller files. The *split* utility performs this task. Having split a file into smaller pieces, the pieces can be edited singly and then can be concatenated into one whole file again with the ***cat*** command.

To split a file *vvs*, containing 100 lines into 25 lines each, we should give the command as:

```
$ split - 25 vvs
xaa
xab
```

xac
xad

Backquotes

The backquote character looks like the single quote or apostrophe, but slants the other way.

It is used to capture the output of a UNIX utility. A command in backquotes is executed and then replaced by the output of the command.

Execute the following commands:
date
save_date= 'date'
echo The date is $save_date

Notice how *echo* prints the output of 'date', and gives the time when you define the *save_date* variable. Store the following in a file named backquotes.sh and execute it (right click and save in a file)

```
#!/bin/sh
# Illustrates using backquotes
# Output of 'date' stored in a variable
Today= 'date'
echo Today is $Today
```

Execute the script with the following command:

sh backquotes.sh

The above example shows how you can write commands into a file and execute the file with a Bourne shell. Backquotes are very useful, but be aware that they slow down a script if you use them hundreds of times. You can save the output of any command with backquotes, but be aware that the results will be reformatted into one line. Try this:
LS=`ls -l`
echo $LS

Example:

Store the following in a file named *simple.sh* and execute it.

```
#!/bin/sh
# Show some useful info at the start of the day
date
echo Good morning $USER
cal
last | head -6
```

After execution, the output shows current date, calendar, and a six previous logins. Notice that the commands themselves are not displayed, only the results.

To display the commands verbatim as they run, execute with

sh -v simple.sh

Another way to display the commands as they run is with -x

sh -x simple.sh

What is the difference between -v and -x? Notice that with -v you see '$USER' but with -x you see your login name. Run the command 'echo $USER' at your terminal prompt and see that the variable $USER stores your login name. With -v or -x (or both) you can easily relate any error message that may appear to the command that generated it.

When an error occurs in a script, the script continues executing at the next command. Verify this by changing ‘cal’ to ‘caal’ to cause an error, and then run the script again. Run the 'caal' script with 'sh -v simple.sh' and with 'sh -x simple.sh' and verify the error message comes from cal. Other standard variable names include: $HOME, $PATH, $PRINTER. Use echo to examine the values of these variables.

Shell Variables

A variable is a name that stores a string. It’s often convenient to store a filename in a variable. Similar to programming languages, the shell provides the user with the ability to define variables and to assign values to them. A shell variable name begins with a letter (upper or lowercase) or underscore character and optionally is followed by a sequence of letters, underscore characters or numeric characters. The shell gives you the capability to define a named variable and assign a value to it.
The syntax is as follows:

$ variable = value

The value assigned to the variable can then be retrieved by preceding the name of the variable with a dollar sign, that is ***$*** variable. For example,

$ length = 50
$ breadth = 20
$ echo $ length, $ breadth

50 20

The “echo” command produces the output in which the values assigned to the variables are printed.

Let us see another example:
$ message = "please log out within 2 minutes"
$ echo $ message
please logout within 2 minutes

The value assigned to a variable can be defined in terms of another shell variable or even defined in terms of itself.
$ length = 50
$ length = display $ length
$ echo $ length
Display 50

The above can be modified into

$ length = 50
$ length = ${length} value
$ echo $ length
50 value

Store the following in a file named *variables.sh* and execute it.

Example:

#!/bin/sh
 # An example with variables
filename="/etc/passwd"
echo "Check the permissions on $filename"
ls -l $filename
echo "Find out how many accounts there are on this system"
wc -l $filename

Now if we change the value of $filename, the change is automatically propagated throughout the entire script.

Performing Arithmetic

If a shell assigned a numeric value, you can perform some basic arithmetic on the value using the command ***expr***. Let us understand with the help of an example.

$ expr 2 + 3
5

expr also supports subtraction, multiplication and integer division. For example:

$ expr 5 - 6
-1
$ expr 11 '' 4*
44
$ expr 5 / 2
2
$ expr 5%2
1
Another example,
$expr 5 * 7

Backslash required in front of '*' since it is a filename wildcard and would be translated by the shell into a list of file names. You can save arithmetic result in a variable. Store the following in a file named *arith.sh* and execute it

Example:

#!/bin/sh
 # Perform some arithmetic
x=24
y=4
Result=`expr $x \* $y`
echo "$x times $y is $Result"

Comparison Functions

The ***test*** command is used to perform comparisons test will perform comparisons on strings as well as numeric values test will return **1** if the conditions is true and **0** if it is false.

There are three types of operations for which *test* are used. There are numeric comparisons, string comparisons and status test for the file system. To compare two values we use flags, that are placed between the two arguments

Test Operators Flag	Meaning
- eq	True if the numbers are equal
- ne	True if the numbers are not equal
- lt	True if the first number is less than the second number
- le	True if the first numbers is less than or equal to the second number
- gt	True if the first number is greater than the second number
-ge	True if the first number is greater than or equal to the second number.

Let us see some examples:

$ test 5 - eq 4
False
The above example returned false because 5 and 4 are not equal.

$ test abc = Abc
False

For string comparison we use = symbol.

Translating Characters

Prepare a text file namely ***vvs.txt*** in which one of the word's is "fantastic". The utility *tr* translates characters

tr 'a' 'Z' < vvs.txt

This example shows how to translate the contents of a variable and display the result on the screen. Store the following in a file named *tr1.sh* and execute it.

Example:

```
#!/bin/sh

# Translate the contents of a variable

name= "fantastic"
echo $name | tr 'a' 'i'
```

Execute the script and see the output.

This example shows how to change the contents of a variable.

Store the following in a file named *tr2.sh* and execute it.

Example:

```
#!/bin/sh

# Illustrates how to change the contents of a variable with tr

name= "fantastic"
echo "name is $name"
name=`echo $name | tr 'a' 'i'`
echo "name has changed to $name"
```

You can also specify ranges of characters. This example converts upper case to lower case

tr 'A-Z' 'a-z' < file

Now you can change the value of the variable and your script has access to the new value

Looping constructs

Executing a sequence of commands on each of several files with for loops

Store the following in a file named *loop1.sh* and execute it.
Note: You have to have three files namely simple.sh , variables.sh and loop1.sh in the current working directory.

Example;

```
#!/bin/sh
# Execute ls and wc on each of several files
# File names listed explicitly
for filename in simple.sh variables.sh loop1.sh
do
        echo "Variable filename is set to $filename..."
        ls -l $filename
        wc -l $filename
done
```

This executes the three commands: echo, ls and wc for each of the three file names. You should see three lines of output for each file name. Filename is a variable, set by "for" statement and referenced as $filename. Now we know how to execute a series of commands on each of several files.

Using File Name Wildcards in For Loops

Store the following in a file named loop2.sh and execute it.

Example:

```
#!/bin/sh
# Execute ls and wc on each of several files
# File names listed using file name wildcards
for filename in *.sh
do
        echo "Variable filename is set to $filename..."
        ls -l $filename
        wc -l $filename
done
```

You should see three lines of output for each file name ending in *.sh*. The file name wildcard pattern **.sh* gets replaced by the list of filenames that exist in the current directory. For another example with filename wildcards try this command:

*echo *.sh*

Search and Replace in Multiple Files

Sed performs global search and replace on a single file.
Note: To execute this example you need to have required files.

sed -e 's/example/EXAMPLE/g' sdsc.txt > sdsc.txt.new

The original file ***sdsc.txt*** is unchanged. How can we arrange to have the original file over-written by the new version? Store the following in a file named s-and-r.sh and execute it.

Example:

```
#!/bin/sh
# Perform a global search and replace on each of several files
# File names listed explicitly
for text_file in sdsc.txt nlanr.txt
do
        echo "Editing file $text_file"
        sed -e 's/example/EXAMPLE/g' $text_file > temp
        mv -f temp $text_file
done
```

First, ***sed*** saves new version in file *temp*. Then, use mv to overwrite original file with new version.

Command-line Arguments

Command-line arguments follow the name of a command. For example:

ls -l .cshrc /etc

The command above has three command-line arguments as shown below:

-l	(an option that requests long directory listing)
.cshrc	(a file name)
/etc	(a directory name)

An example with file name wildcards:
*wc *.sh*

How many command-line arguments were given to wc? It depends on how many files in the current directory match the pattern ****.sh***. Use 'echo *.sh' to see them. Most

UNIX commands take command-line arguments. Your scripts may also have arguments.

Store the following in a file named *args1.sh*

Example:

```
#!/bin/sh
# Illustrates using command-line arguments
# Execute with
#             sh args1.sh On the Waterfront
echo "First command-line argument is: $1"
echo "Third argument is: $3"
echo "Number of arguments is: $#"
echo "The entire list of arguments is: $*"
```

Execute the script with

sh args1.sh -x On the Waterfront

Words after the script name are command-line arguments. Arguments are usually options like *-l* or *file names.*

Looping Over the Command-line Arguments

Store the following in a file named args2.sh and execute it.

Example:

```
#!/bin/sh
# Loop over the command-line arguments
# Execute with
#          sh args2.sh simple.sh variables.sh
for filename in "$@"
do
          echo "Examining file $filename"
          wc -l $filename
done
```

This script runs properly with any number of arguments, including zero. The shorter form of the *for* statement shown below does exactly the same thing:

```
for filename
do
...
```

But, don't use:

*for filename in $**

It will fail if any arguments include spaces. Also, don’t forget the double quotes around $@.

***If* construct and *read* command**

Read one line from ***stdin***, store line in a variable.

read variable_name

Ask the user if he wants to exit the script. Store the following in a file named ***read.sh*** and execute it.

Example:

```
#!/bin/sh
```

```
# Shows how to read a line from stdin
echo "Would you like to exit this script now?"
read answer
if [ "$answer" = y ]
then
        echo "Exiting..."
        exit 0
fi
```

Command *Exit* Status

Every command in UNIX should return an exit status. Status is in range 0-255.Only **0** means success. Other statuses indicate various types of failures. Status does not print on screen, but is available through variable $?.

The following example shows how to examine exit status of a command.

Example:

Store the following in a file named ***exit-status.sh*** and execute it.

```
#!/bin/sh
# Experiment with command exit status
echo “The next command should fail and return a status greater than zero”
ls /nosuchdirectory
echo “Status is $? from command: ls /nosuchdirectory”
echo “The next command should succeed and return a status equal to zero”
ls /tmp
echo “Status is $? from command: ls /tmp”
```

Example given below shows if block using exit status to force exit on failure.

Store the following in a file named ***exit-status-test.sh*** and execute it.

Example:

```
#!/bin/sh
# Use an if block to determine if a command succeeded
echo "This mkdir command fails unless you are root:"
mkdir /no_way
if [ "$?" -ne 0 ]
then
        # Complain and quit
        echo "Could not create directory /no_way...quitting"
        exit 1  # Set script's exit status to 1
fi
echo "Created directory /no_way"
```

Regular Expressions

For searching zero or more characters we use the wild characters.*. Let’s see the examples:

```
grep ‘provided.*access’ sdsc.txt
sed -e ‘s/provided.*access/provided access/’ sdsc.txt
```

To search for text at beginning of line we give the command,

```
grep '^the' sdsc.txt
```

To search for text at the end of line we give the command,

```
grep 'of$' sdsc.txt
```

Asterisk means zero or more the preceding characters.

a* zero or more a's
aa* one or more a's
aaa* two or more a's

Examples:
Delete all spaces at the ends of lines
sed -e 's/ *$//' sdsc.txt > sdsc.txt.new

Turn each line into a shell comment
sed -e 's/^/# /' sdsc.txt

The *case* statement

The next example shows how to use a *case* statement to handle several contingencies. The user is expected to type one of three words. A different action is taken for each choice.

Store the following in a file named ***case1.sh*** and execute it.

Example:

```
#!/bin/sh
# An example with the case statement
# Reads a command from the user and processes it
echo "Enter your command (who, list, or cal)"
read command
case "$command" in
        who)
                echo "Running who..."
                who
                ;;
        list)
                echo "Running ls..."
                ls
                ;;
        cal)
                echo "Running cal..."
                cal
                ;;
        *)
                echo "Bad command, your choices are: who, list, or cal"
                ;;
esac
exit 0
```

The last case above is the default, which corresponds to an unrecognised entry. The next example uses the first command-line arg instead of asking the user to type a command.

Store the following in a file named ***case2.sh*** and execute it.

Example:

```
#!/bin/sh
# An example with the case statement
# Reads a command from the user and processes it
# Execute with one of
#       sh case2.sh who
#       sh case2.sh ls
#       sh case2.sh cal
echo "Took command from the argument list: '$1'"
case "$1" in
        who)
                echo "Running who..."
```

```
                who
                ;;
        list)
                echo "Running ls..."
                ls
                ;;
        cal)
                echo "Running cal..."
                cal
                ;;
        *)
                echo "Bad command, your choices are: who, list, or cal"
                ;;
esac
```

The patterns in the case statement may use file name wildcards.

The *while* statement

Example given below loops over two statements as long as the variable *i* is less than or equal to ten. Store the following in a file named while1.sh and execute it.

Example:

```
#!/bin/sh
# Illustrates implementing a counter with a while loop
# Notice how we increment the counter with expr in backquotes
i="1"
while [ $i -le 10 ]
do
        echo "i is $i"
        i=`expr $i + 1`
done
```

The example given below uses a *while* loop to read an entire file. The *while* loop exits when the read command returns false exit status (end of file). Store the following in a file named ***while2.sh*** and execute it.

Example:

```
#!/bin/sh
# Illustrates use of a while loop to read a file
cat while2.data |   \
while read line
do
        echo "Found line: $line"
done
```

The entire *while* loop reads its *stdin* from the pipe. Each *read* command reads another line from the file coming from *cat*. The entire *while* loop runs in a subshell because of the pipe. Variable values set inside while loop not available after *while* loop.