UNIT 2 SHELL PROGRAMMING

Structure

2.0 Introduction
2.1 Objectives
2.2 Programming in The Bourne and the C-Shell
2.3 Wild Cards
2.4 Simple Shell Programs
2.5 Variables
2.6 Programming Constructs
2.7 Interactive Shell Scripts
2.8 Advanced Features
2.9 Summary
2.10 Model Answers

2.0 INTRODUCTION

The Shell in UNIX is a wonderful entity that serves us in various ways. It is started up automatically every time you login to the system. The Shell sets up your environment when you start off on the machine. It is the Shell that lets you run different commands without having to type the full pathname to them, even when they do not exist in the current directory. The Shell expands wildcard characters, thus saving you laborious typing. It gives you the ability to run previously run commands without having to type the full command again. It is the Shell that does input, output and error redirection.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- discuss various features of shell programming constructs
- List several wild card characters

2.2 PROGRAMMING IN THE BOURNE AND THE C-SHELL

You can use the Shell as a programming language. The C- Shell has all the usual language constructs like sequencing, looping, decisions, variables, functions and parameters. In this section we will look at these abilities of the Shell, although regretfully not in complete detail. To look closely at the C- Shell's capabilities would require a block in itself. That is why as usual we will only take a few portions in detail and leave you to the documentation for the rest. As with any other programming language, the key to mastering the C-Shell is practice. Keep writing Shell programs whenever needed and you will soon feel comfortable with them.

When you have written some complex Shell programs you will realise that some UNIX utilities which you might have thought were useless, are actually very important and useful. Sed is a good example of such a utility.

Shell programs are very often called shell scripts. By the way a Shell is a program like any other, which means that there can be different shells. Actually there are some other popular shells, of which the Bourne shell and the Korn shell are quite likely to be available at your installation. The Bourne shell was the first shell, in fact, and is sure to be there on your machine.

If you look at a UNIX machine, you can think of it as being composed of several layers. At the lowest layer is the hardware which does all the physical tasks and without which there would be no computer and no UNIX. Above that is the UNIX kernel which is the core of the operating system and does memory management, device handling and all the other mundane tasks needed to make the hardware easily usable by us. The UNIX commands and utilities

come next. At the top is the Shell which can be considered to be the outermost layer and which enables us to run the utilities and other UNIX commands. You can construct higher application layers of your own which run above the Shell. Although we have talked of the Shell as being a different layer, it is as much a program as any of the UNIX utilities. For that matter even the kernel is a set of programs.

We have been referring to the Shell's activities in the previous units, mainly when talking of wild cards. We start our description of the C-Shell with this subject.

While the C-Shell is very useful and has many features, there are some things that are easier to do in the Bourne shell. Moreover many system and third party shell scripts are written in the Bourne shell. So in this unit we will look at both these shells.

2.3 WILD CARDS

Wild card characters are characters which can stand for characters other than themselves, somewhat like a joker in a pack of cards (the joker has no intrinsic meaning by itself). A judicious use of wild card characters can make many commands easy to issue by saving a lot of typing and preliminary research.

Suppose you have been writing a series of programs for enciphering text. You have been calling them cph01.C, cph02.C and so on. You suddenly realise that you have been doing this in the directory ~/khanz/crypt, while actually these programs are for a particular project and you would like them in the directory ~/khanz/crypt/knapsack. All you have to do to rectify the situation is to move your programs to the correct directory after creating it. So you start off

```
% cd ~/khanz/crypt
% mkdir knapsack
% mv cph01.C knapsack
% mv cph02.C knapsack
% mv cph03.C knapsack
```

Soon you get sick of typing almost the same thing again and again, which can easily happen if you have just 19 programs. You could stop as soon as mv reported a non-existent source file, but if you had mistakenly left gaps while naming the files, you could not be sure that there were no more files. To be sure, you would have to do an ls to look at the directory listing first. Then you would have to keep issuing mv commands until you succeeded in moving all the files. Here you would probably need to keep checking the directory listing from time to time.

It is much less effort if you use wild cards. Just say the following after you have made the sub-directory you want

```
% mv cph??.C knapsack
```

The ? is a wild card character and can stand for any single character including itself, much like the . in vi. So cph??.C expands to cph followed by any two characters, followed by .C. This is quite likely all you need to say. If it does not work, it can only be because you named some of the programs differently, thus invalidating our basic assumption, not because of a problem with wild cards.

The ? metacharacter implies the existence of exactly one character in that position. So cph1.C will not be matched by the command above. You need to remember this. If, for example, you had named the files cph1.C, cph2.C, ..., cph10.C, cph11.C and so on, then the command

```
% mv cph??.C knapsack
```

will leave any files from cph1.C to cph9.C in the same directory as before. You can move them too with ease

```
% mv cph?.C knapsack
```

But what if your naming convention only started the filenames with cph and allowed any

number of characters after that (this is quite common)? Is there no easy way out? Even assuming a limit of 14 characters in the filename (some systems do not have a limit), and counting 2 fixed characters at the end and 3 at the beginning, you are left with upto 9 characters to look for in the middle. So you can use the following method

```
% mv cph.C knapsack
% mv cph?.C knapsack
% mv cph??.C knapsack
% mv cph???.C knapsack
% mv cph????.C knapsack
% mv cph?????.C knapsack
% mv cph??????.C knapsack
% mv cph???????.C knapsack
% mv cph????????.C knapsack
% mv cph?????????.C knapsack
```

While this is much simpler than looking at the directory listing and then issuing mv commands, it entails some labour anyway. This was the time to say

```
% mv cph*.C knapsack
```

You have now seen the * metacharacter, which matches any sequence of characters in that position, including the situation where there is no character at all. So the * here matches from 0 to 9 characters (more if required), and those characters can be any character including the * itself, except a leading period.

You cannot at this stage wonder how to put a ? or a * in a filename, except that in the Bourne shell a command like

```
$ vi cph*.C
```

is not expanded unless there is at least one filename around which satisfies the expansion of the command. So if there is a directory which does not have any file whose name starts with cph, then a filename cph*.C is created, where the * is taken literally. In other cases, you can create such a file by

```
% vi cph\*.C
```

escaping the * with a \ as usual. A ? can be put into a filename in the same way. But it would be a good idea to avoid such characters in filenames so that there is no confusion.

Check your Progress 1

1. Try the command

   ```
   % ls *
   ```

   What filenames does it neglect to furnish you with? How can you get all filenames?

   .....

   .....

2. Try the above command in a directory which has a large number of files. What happens when you have, say 500 files? What then is the difference between ls and ls * and why? If you experiment you will find a situation where

   ```
   % cd /usr/khanz/prj/crypt/src
   % ls *
   ```

   works but a command like

   ```
   % ls /usr/khanz/prj/crypt/src/*
   ```

   fails. Why does this happen?

   .....

.....

.....

There is a UNIX command called echo which simply echoes or lists its arguments. For example

% echo Hello world I am learning C

Hello world I am learning C

How can you use this command to get a directory listing?

.....

.....

2.4 SIMPLE SHELL PROGRAMS

We spoke earlier about how we could use the C-Shell as a programming language. Before we see how to write programs using all the capabilities of the Shell, we will look at some simple shell scripts to illustrate the concept.

Suppose you, khanz, are working on your cryptography project in a directory ~/prj/crypt/pkc/src where your source files are located. This is where you are most of the time. But sometimes you have to stray away from here to look at documentation which you have stored elsewhere, or to some other project you are working on. It is cumbersome to type out the whole pathname of this directory to search it. Let us create a shell script in your home directory to do the job. Use vi to create a file wd

% cat wd

ls ~/prj/crypt/pkc/src

Now go to some distant directory like /usr/include/sys and type

% wd

Nothing happens, or rather you are told that wd cannot execute. This is because you do not have execute permission on the newly created file wd (unless you have set the umask to allow it). So we need to acquire this permission, which is easy because we own the file

% chmod 755 ~/wd

Now you can try typing ~/wd to execute the file. You could also have tried

% csh ~/wd

which would also have executed. We have looked at this in the discussion of the mail command, and we will look at in some more detail later.

Note that if you just type

% wd

in a distant directory, you will not be able to execute the command. This is because when the Shell is given a command to execute, it looks for the command in different directories in a fixed sequence. The first directory searched is the current directory, followed by /bin and /usr/bin. These are only typical settings and can be controlled by you in a manner which will be soon explained. Since wd is not in any of these directories, UNIX does not find the command and complains. Until you learn how to fix this, you can continue with ~/wd.

If you consider it carefully, you will realise that you have now created a UNIX command of your own. If you give others execute permission on the file, anybody else could type wd and search your directory. While that is probably not what you would want to encourage, you could create more generally useful shell scripts presently. This is at the core of the UNIX philosophy of building on the work of others. Once somebody has written a command, you

do not have to write it again. But if you want to do something slightly different, you can probably use the available command and use other commands to modify its effect. We will see examples of this by and by.

You have to be careful when you write generally useful shell scripts. While you can afford to be a wee bit sloppy when you are writing for yourself, and can sometimes leave weaknesses in the command because you know of them and will take care not to use the command in those situations, you cannot release a command for public use so casually. Most installations have a set of shell scripts and programs in the usual programming languages available. These are locally developed or acquired programs which are found to be useful at that site. You should learn to use them to advantage before you try to write your own commands for some task. Such utilities are usually placed in /usr/local/bin. If you have special purpose utilities that are not generalised enough to go there, you can collect them in your directory tree in a place like ~/bin.

Let us now look at some of the pitfalls a novice might come across when writing his own shell scripts. Suppose you want to have your ls command work like ls -al and you do not want to type out the full command every time. So in your home directory you can create a file ls

```
% cat ls
ls -al
```

and make it executable. Now when you try to look at the directory listing

```
% ls
```

you get a blank look from UNIX and you will not need to interrupt the command. This is because the command is calling itself in an infinitely recursive fashion. You will therefore need to change your command

```
% cat ls
/bin/ls -al
```

so that it calls the real ls. That is how you can hope to get any work done, because your ls relies on the system's ls command to get you the directory listing.

This will now work, but not if you try a command like

```
% ls prj/crypt
```

You will still get the listing of your current directory. That is because we have not given our ls the ability to recognise command line arguments. We will soon see how to do that. Until then, our ls is not going to be very useful.

Check Your Progress 2

1. Write a shell script Cat that prints out the contents of some fixed file in upper case.

   .....

   .....

2. Write a shell script words that prints out a list of every unique word contained in the file in alphabetical order. You should here use the work you did in the previous unit.

   .....

   .....

2.5 VARIABLES

We now come to the basic features of the Shell programming language. Let us see how variables can be defined and used in the C-Shell. In the course of this discussion we will solve many other mysteries which we have been putting off till now, like how to make UNIX

recognise a command in some directory, how to change the UNIX prompt, how to customise your environment when you login and several other things.

It is easy to define and set a variable in the C-Shell and look at its value. For example, you can try

```
% set vehicle=bus
```

This creates a Shell variable called vehicle and sets its value to "bus". To look at the value say

```
% echo $vehicle
```

```
bus
```

If we have another variable called vehicle1 with a value car, we can say

```
% echo $vehicle and $vehicle1
```

```
bus and car
```

The echo command has been given three arguments here of which the first and third are shell variables and the second is a constant string. The command dutifully prints out the values of the variables and the constant is unchanged.

But if you want to now define another variable adj to be "business", then you cannot say

```
% set adj=$vehicleiness
```

```
% echo $adj
```

Since there is no variable called vehicleiness, this sets the value of adj to a null string. You can achieve the effect you want by enclosing the available variable in braces

```
% set adj=${vehicle}iness
```

```
% echo $adj
```

```
business
```

If a variable is to be set to a string containing spaces, then the string needs to be enclosed in quotes, single or double

```
% set greeting="How do you do"
```

```
% echo $greeting
```

```
How do you do
```

Once you have defined a variable you can freely use its value. So if you have said

```
% set Cat="cat ~/.exrc"
```

then you can always print out your .exrc file by saying

```
% $Cat
```

```
set numberset tabstop=4
```

```
set terse
```

Another way of setting Shell variables is by making them equal to the output of a command. To use the output of a command in the Shell you have to put the command in the accent grave, or back quotes.

Similarly you could set a variable to the current directory

```
% set dir=`pwd`
```

```
% echo $dir
```

```
/usr/khanz/prj/crypt/src
```

We will now see how to pass arguments to Shell scripts so that they are not ignored when the command is executed. You have seen how the private ls command you had created did not take any arguments into consideration. The command was therefore not of much use. You will now learn to rectify this shortcoming. In fact many of the other toy Shell scripts we have written so far are all but useless if we cannot get them to accept arguments and act on them. For example the Cat command given as an exercise would be much more useful if one could ask it to print the file of one's choice in upper case rather than some fixed file.

You can make a shell script refer to command line arguments by using the notation Sn for the nth argument. Only 9 arguments can be so accessed. These are also called positional parameters because they are referred to by their position on the command line instead of by name. The name of the shell script itself can be referred to as $0. So you can write an improved version of your ls command

```
% cat ~/ls

/bin/ls -al $1
```

This version will accept the first argument given to it. So if you say

```
% ~/ls /usr/khanz/prj/crypt/src
```

you will get a long listing of all files in that directory. But even this fails if you give a second argument or more arguments. Try the following command

```
% ~/ls /usr/khanz/prj /usr/khanz/doc
```

Now only the first argument is recognised and the second is ignored. Well, let us have an ls which recognises 9 arguments. So we get into vi and change our command to

```
% cat ~/ls

/bin/ls -al $1 $2 $3 $4 $5 $6 $7 $8 $9
```

But what if there are more than 9 arguments? Are we doomed to have only such a limited version of our ls? Fortunately in this case you can easily solve the difficulty by using the notation $* which stands for all the arguments, whatever the number. This way we can get around the limitation of being able to handle only 9 positional arguments. Thus we now have a full blown ls command of our own

```
% cat ~/ls

/bin/ls -al $*
```

We can also find out the number of arguments by saying $#. This excludes $0, the name of the shell script itself. So you can write a simple command to count the number of arguments given to it. Such a capability is also useful when you want to write production shell scripts which need to take care of error checking, for instance, ensuring that the shell script works only if it is called with the proper number of arguments, or even performing a different action depending on the number of arguments passed to it.

```
% cat count

echo $#

% count apple boy cat dog elephant fish

6
```

There is also a shift command which has the effect of shifting its arguments such that each positional parameter gets the value of the next one, and the original value of the first parameter is discarded. Using this facility you can deal with a situation where there are more that 9 positional arguments. Let us see a simple example

```
% cat showargs
echo "\$1 = $1"
echo "\$2 = $2"
echo "\$3 = $3"
```

```
echo "\$4 = $4"
echo "\$5 = $5"
echo "\$6 = $6"
echo "\$7 = $7"
echo "\$8 = $8"
echo "\$9 = $9"
echo "\$10 = $10"
```

The echo command can be given the -c argument to suppress the printing of the newline character, -n for inserting a newline character, -f for form feeds and -t for tab characters. In the C- Shell the -n option suppresses the printing of the newline. Any variables enclosed in double quotes are substituted for their actual values, but anything in single quotes is not altered. If you want a character literally within double quotes, you can use a backslash to escape it. Within single quotes you would not need the backslash, but then you would not be able to get the values. Quotes of one kind can be used to enclose quotes of the other kind. Now we can understand the output of showargs

```
% showargs apple boy cat dog elephant fish girl house inkpot jug
    $1 = apple
    $2 = boy
    $3 = cat
    $4 = dog
    $5 = elephant
    $6 = fish
    $7 = girl
    $8 = house
    $9 = inkpot
    $10 = apple0
```

You can see that $10 has been interpreted as $1 followed by the literal 0. To take care of this we will alter showargs (only the last lines are shown)

```
% cat showargs
...
echo "\$9 = $9"
shift
echo "\$10 = $9"
```

Now the output of the previous command changes to the following

```
$9 = inkpot
$10 = jug
```

There are some variables in the Shell which are predefined, which means that their values do not have to be set by you. When you login those variables are created and set appropriately. However many of them can be altered subsequently. You can see a list of all variables currently defined together with their values by saying

```
% env
```

printenv has the same effect. You should check out your environment with the env command. You can easily make out the meanings of most of them because the names are suggestive of the meaning. Thus LOGNAME is your login name, HOME is your home directory and so on.

You can also use the set command to display the values of all variables set in the current shell. For example the default C-Shell prompt is the percentage sign %, stored in the shell

variable prompt. This is the key to customizing your environment when you login, as we have always been promising you could.

In the C-Shell every time you login, UNIX looks for a file in your home directory called ".login". If this file exists then all commands in that file are executed before you are presented with your first system prompt. There is a corresponding file which is executed when you log out of the system. This file is called ".logout".

Suppose you usually work in your /usr/khanz/prj/crypt/c++/src directory. You would like to be placed in this directory automatically every time you login. All you have to do is to put the following line in your .login file

```
cd prj/crypt/c++/src
```

Since you will anyway reach your home directory, we have given a relative pathname to reach the desired directory. If you want your UNIX prompt to give you the number of the command you will be executing, you can say

```
set prompt=\!\%\
```

The exclamation mark stands for the command number and we escape it to prevent it from being evaluated immediately, which would make the prompt constant. Now you see a prompt like

```
1% pwd
/usr/khanz/prj/crypt/c++/src
2%
```

Every time a command is executed the command number increases by one and this shows up on your prompt. Commands separated by semicolons are considered to be one command, as you know.

There are some other convenient features which can be set up for a pleasing environment. Of course these are a matter of individual taste, so we will talk of those done by most people. You can always choose what you feel congenial for yourself. If you say

```
% set ignoreeof
```

then you will have to type out "exit" or "logout" in order to terminate your login session, a mere ^D will result in a message

```
Use "logout" to logout
```

This is useful to prevent accidental log outs because ^D can get typed by mistake while it is much more unlikely you will type a full word when you do not need it. Another safety feature is to say

```
% set noclobber
```

If this variable is set then an operation which would otherwise overwrite an existing file silently will refuse to do with the message

```
file exists
```

Both these variables need only be set, and no values need to be put in them. The C-shell distinguishes between variables whose value is null and those not defined at all.

A very convenient feature of the C-Shell (this is absent in the Bourne Shell) is the facility of a command history. If you set the variable history then the last commands you typed in can be executed very easily without your having to type the complete command in

```
% set history=100
```

will allow you to access the last 100 commands you typed. If you want to see all of them, just say

```
% history
```

and the last 100 commands will be displayed. You can set history to a smaller value of about 20, this is probably sufficient most of the time.

If history is set, you can run a previous command in two ways. You can say

```
% !56
```

and the 56th command of your session (not 56th in the current history) will be executed. This includes all command line arguments, redirections, pipes and all other special characters it might have contained. Usually when you want to repeat some command you do not know its number, but you of course know what the command was. Let us say you ran a longish sed command to perform some action on a file and now you need to do it again after some change has occurred to the input. You need to say

```
% !sed
```

and the last command starting with sed will run. You do not have to give the whole word sed, any number of characters which make the command unambiguous will do. So if you have not run any command starting with se after you ran sed, you can say

```
% !se
```

and if you have not run any command with s itself, you could have abbreviated it still further. But remember that the last command is the one that will be run. If there were a command like sedate and if you had run it after sed, then the above abbreviated commands would have run it instead of sed.

There are ways of editing the command before running it but it is usually simpler to retype it again in that case. Also the abbreviated command can be used to run another command containing it like this

```
34% sort datafile

...

48% !34  newfile

sort datafile  newfile
```

There is one aspect of the Shell which we have mentioned earlier but which needs elaboration. Whenever a command is executed it does so in a child shell which inherits the environment of the parent shell. If you want variables you have set to have effect in child shells as well, you need to say setenv rather than just set. Otherwise those variables will not be passed onto child shells. You can easily try this out.

Another thing is that you cannot make a change to your .login file, run .login and have the changed environment. This does not mean that the commands are not run. The point is that they are run in a child shell and as soon as the command terminates, you are back in the parent shell where there is no effect of anything that happens in a child shell. The environment is passed downwards to child shells but cannot be passed back to parent shells. To make a change to your .login and have it take effect immediately, you should say

```
% source .login
```

This runs the .login file exactly as if you had typed out each command at the prompt instead of executing the commands in a child shell. To be precise, the source command temporarily (for the duration of the command) redirects the standard input of the shell to come from the file instead of from the terminal. This command can of course be given any shell script as an argument and is not confined to .login.

One more feature which is sometimes useful in the C-shell is the repeat command. You can run a command several times repeatedly by saying

```
% repeat 17 echo hello
```

and you will find that the echo is executed 17 times.

You can use the alias feature of the C-shell to abbreviate commands (strictly to give them another name). Thus if you frequently use ls -al, you can say something like

```
% alias ll ls -al
```

Now whenever you want to use ls -al, just say ll and you will get the same effect. You can give arguments to ll and they will be taken as additional arguments to ls -al, the actual command. To turn this feature off, just say

```
% unalias ll
```

Usually such aliased commands are placed in one's .login file so that they are in effect automatically as soon as one logs in.

2.6 PROGRAMMING CONSTRUCTS

So far the shell scripts we have written have been nothing but a sequence of commands which we could very well have given from the system prompt itself in most cases. They would have been useful but not very much so, if the capabilities of the Shell were confined to setting variables and printing their values. What makes the shell powerful is that we can use it as a programming language. It has all the constructs needed to write a program, and in this section we shall look at them.

While the C-Shell is very convenient to use on the command line because of its greater set of features suitable for interactive use, the Bourne shell is sometimes more convenient in some ways when you want to write a shell script. For this reason in this section we will discuss mainly the Bourne Shell and will give the C-Shell features sometimes, because of lack of space. As always, you are regretfully referred to the UNIX documentation for the details of how to program in the C-shell. But remember that even if you use the C-shell for interactive commands, you can always run a Bourne shell by giving a : by itself on the first line of the shell script. This is also the default shell used for shell scripts. So even if you omit the : the script will be assumed to be a Bourne shell script and will be run as such by default. You can run a C-shell script by putting a # instead of a : on the first line.

In the following examples you can distinguish between the Bourne and the C-shell scripts by the prompts used ($ for the Bourne shell and % for the C-shell), and by the : or the # in the scripts.

Let us first look at the looping constructs available. There is a for loop which works like this in the Bourne shell, which has a default $ prompt

```
$ cat mkupper
:
for i in `ls`
do
tr '[a-z]' '[A-Z]' $i $i.up
done
```

The C-shell version of the same script will be like this

```
% cat mkupper
#
foreach i (`ls`)
tr '[a-z]' '[A-Z]' $i $i.up
end
```

This script will change every file in the current directory to upper case and write it out with a .up extension. The i is a variable we use to give effect to our plans, and it serves as a loop variable. The ls command can have several values and the loop is executed for each of the values in turn. For every value, the file is converted to upper case and written out with a ".up" extension. When all the files have been processed, the loop terminates. We can write another version of this script which will convert only the files specified as arguments to it into upper case.

```
$ cat mkupper.arg
```

```
for i in $*
do
tr '[a-z]' '[A-Z]' $i  $i.up
done
```

Here we can give the script any number of arguments and every file we specify will be converted into upper case. Actually in this case we can omit the in and the first line can be

```
for i
```

which will work just as the previous line. One can also execute loops with the while or until constructs, which have the usual meanings.

```
$ cat mkupper.1
:
while test $# -gt 0
do
tr '[a-z]' '[A-Z]' $1  $1.up
shift
done
```

Here the while loop is performed as long as there are some arguments left. The tr operation is done only on the first argument every time. The shift then renumbers the arguments and throws out the first argument (which has already been dealt with). When there are no arguments left the loop terminates. The C-shell has its own version of the while loop.

The until loop is exited only when the condition becomes true. This construct is not available in the C-shell, but you can get around that quite easily by negating the condition of a while loop. Let us look at the version with this loop construct

```
$ cat mkupper.2
:
until test $# -eq 0
do
tr '[a-z]' '[A-Z]' $1  $1.up
shift
done
```

Now the loop is terminated when the number of arguments is 0. Of course since we have changed the condition to be tested, the effect of this is the same as in the while loop case.

In the looping examples above, we have been using the test operation, which can be used in the Bourne shell to test various conditions. The operation returns a 0 if the test is passed and a non zero value otherwise. We will quickly describe the conditions which can be tested. Let us first look at the relational operators on positive or negative integers. You can say

test a -gt b (for checking if a b)

test a -ge b (for checking if a = b)

test a -eq b (for checking if a = b)

test a -ne b (for checking if a != b)

test a -lt b (for checking if a b)

test a -le b (for checking if a b)

In the translation examples above, we have used this numeric test to stop the loop when there are no arguments left. Suppose there were no arguments given to the command mkupper when it was run. One could exit silently, or print an error message, or use the standard input in such a case. The examples given all exit silently if there are no arguments. Let us see how to change this behaviour.

```
$ cat mkupper.3
:
if test $# -eq 0
then echo "No files to translate!"
exit
fi
while test $# -gt 0
do
tr '[a-z]' '[A-Z]' $1 $1.up
shift
done
```

The beginning of the script introduces to the if statement, which is how a decision can be made in the shell. We will look at this statement and its variants a bit later. The fi delimits the statements to be performed if the condition is true. This script will tell you if are no files to translate and will put you back to the shell prompt if that happens. You can now easily write a script which takes input from the standard input if no arguments are supplied to the command when it is invoked.

We can now look at some of the tests that can be performed on files. Here you can use ! as the negation operator to check for the inverse of a condition.

test -s filename (does a non-empty filename exist?)

test -f filename (is filename an ordinary file?)

test -d filename (is filename a directory?)

test -r filename (is filename readable?)

test -w filename (is filename writeable?)

The tests for reading and writing are for the user running the script, that is, you can find out if you can read from or write to the file in question. Using these tests, we can make our script more robust by checking if the files given as arguments exist

```
$ cat mkupper.4
:
if test $# -eq 0
then echo "No files to translate!"
exit
fi
while test $# -gt 0
do
if test -s $1
then if test -f $1
then tr '[a-z]' '[A-Z]' $1 $1.up
fi
fi
shift
done
```

Here we have used two if statements within the while loop to check for the existence of each file given as an argument to the command. You should expand this script to print an appropriate error message whenever needed. Notice that the actual action we are performing is confined to the one line tr command. It is the error checking that is now taking up more and more of the script. This is in keeping with what happens in most programs, where error checking takes up a large part of the production version of any software.

You can also compare two character strings as follows

test a = b (is string a the same as string b?)

test a != b (are the strings different?)

You can also use test -z which is true if a string is null, and test -n which is true if the string is not null. Also, test by itself, if followed by a string, checks if the string exists. You should now rewrite the previous script to check for the absence of an argument using the appropriate form of the test operation.

You can combine two tests with -a for an and operation (true if both the tests pass) or -o for an or operation (true if either of the tests pass). This can be used to rewrite the scripts above with only one if statement.

The if statement itself is the same as in conventional programming languages. If the condition following the if keyword is true the statements that follow are executed. The body of the if is terminated with a fi. Suppose you want to write a script that will take three arguments. The first two should be filenames that exist and the third is to be a new file that will hold the concatenation of the upper case versions of the first two files. We thus need to ensure that the first two filenames exist and the third does not.

The if statement can have an else part which is executed if the condition following the if is not true. In such a case the then is delimited by the else and the whole statement is delimited by the fi. The else if construct is also available and can be combined as an elif keyword. In the C-Shell, the delimiter is the keyword endif and you can give any number of else parts.

```
S cat Concat
:
if test S# -ne 3
then echo "Usage: $0 in-fl1 in-fl2 ou-fl"
exit
elif test ! -s $1 -o ! -f $1
then echo "Invalid filename $1"
exit
elif test ! -s $2 -o ! -f $2
then echo "Invalid filename $2"
exit
elif test -s $3 -o -f $3
then echo "$3 exists"
exit
else tr '[a-z]' '[A-Z]' $1  tmp1
tr '[a-z]' '[A-Z]' $2  tmp2
cat tmp1 tmp2  $3
rm tmp1 tmp2
fi
```

The break and continue statements are available with loops to come out of the loop and go back to the beginning of the loop respectively.

We can also put comments in the Bourne Shell by using a #. All statements on a line following the # including the # itself are ignored by the shell during execution. In complex shell scripts it is a good idea to include comments explaining what the script does and how it works.

In the example above we used a multiple if statement. In some situations it is more convenient to use the case statement which is illustrated below. The example is a trivial one but does show how the statement works

```
S cat showcase
case $2 in
```

```
upper) tr '[a-z]' '[A-Z]' $1 $1.up;;
lower) tr '[A-Z]' '[a-z]' $1 $1.low;;
*) echo "Invalid option";;
esac
```

This script takes two arguments. The first is a filename which is converted into upper or lower case depending on the second argument. The value of the second argument is checked against the strings upper and lower. These strings are delimited by a right parenthesis character). After this is the body of statements that will be executed if a match is found. The shell metacharacters can be used as strings to match against. The whole case statement is terminated with a esac and the action for each option is ended with two semicolons.

Now you should try the following variation of the script

```
$ cat showcase.1
case $2 in
*) echo "Invalid option";;
upper) tr '[a-z]' '[A-Z]' $1 $1.up;;
lower) tr '[A-Z]' '[a-z]' $1 $1.low;;
esac
```

This is the same as the earlier one except that the * now appears first in the list of strings to check against. Since any string will match this, this is the action that will be taken every time. Thus it is the first match found which is executed in a case statement. So the order in which you write the matches in the body of the case is very important.

2.7 INTERACTIVE SHELL SCRIPTS

So far we have only looked at shell scripts which have not required any interaction with the user once started. But the power of the shell does not stop at the constructs we have seen so far. It is possible to take input from the user and take further action depending on it. You can thus write interactive programs.

To pass user input to a command inside a shell script, we can follow the command with the notation &0. Although this is not necessary and user input will usually get passed to the command inside the script, this notation ensures that this will happen. Thus if in a shell script it is necessary to call up vi in read only mode on some files (to allow the user to examine them) you will want that the user be able to use the view commands on those files. For this put the line which calls up vi like this

```
view $* &0
```

If you only want to accept input from the terminal directly for the shell program, you say

```
read x
```

and whatever you type will get assigned to the variable x.

In one read statement you can have several variables to which you can assign values. The first response is assigned to the first variable, the second response to the second variable, and so on. Responses are delimited by white space but a newline does the assignment. If there are too many responses, all the extra responses go into the last variable. Using these features you can write a menu driven shell program to perform some tasks interactively. As you have seen, UNIX commands are cryptic and do not have a helpful user interface. You could write a menu driven ls program, for instance. Let us call it dir.

```
$ cat dir
echo "1 for long listing"
echo "2 for stream list"
echo "3 for single column list"
echo "Enter your choice \c"
read x
```

```
case $x in
1) ls -l $*;;
2) ls -m $*;;
3) ls -1 $*;;
*) echo "Invalid choice"
esac
```

This script has a long way to go before you could release it as a useful utility for beginners, though. There is no error checking beyond looking for a correct choice. You can try making this a bit more robust. For instance, if the user gives two input options, what will happen?

This seems like a good place to be talking of redirection. As you have seen in the introductory unit of this block, you can redirect the standard output of a command to another device or file. In the C-shell which we had been looking at there, a command like

```
% ls -l  dirfile
```

redirects the standard output to the file specified, dirfile in this case. The standard error, which is not being redirected, is sent to the terminal screen as usual. To send the standard error also to errfile we have to say

```
% ls -l  &errfile
```

Here both the standard output and standard error are sent to errfile. The task of sending the standard output and standard error to different files is slightly more complicated, as we have to use sub-shells for this. Enclosing any command in parentheses executes that command in a sub-shell and as soon as the command terminates you are back in the parent shell. We can thus redirect the standard output to dirfile and the standard error to errfile at the same time as follows

```
% (ls -l  dirfile)  &errfile
```

In the Bourne shell, on the other hand. things are much more straightforward. To redirect the standard output alone, say

```
$ ls -l  dirfile
```

This sends the standard output to dirfile and the standard error to the terminal, as usual. To redirect the standard error, say

```
$ ls -l 2 dirfile
```

Now the standard output will appear on the terminal and the standard error will get redirected. This is because in the Bourne shell you can specify the file descriptor to be redirected by placing it before the sign. The standard input is assigned to file descriptor 0, the standard output to 1 and the standard error is assigned to file descriptor 2 automatically by UNIX. It should now be easy for you to redirect the standard output and the standard error of a command to different files.

```
$ ls -l  dirfile 2 errfile
```

This command sends the standard output to dirfile and the standard error of the same command to errfile. To redirect both of them to the same file you will need to learn the idiom

```
$ ls -l  dirfile 2&1
```

which sends the standard error to the same place as the standard output.

You have already seen in the first unit that in UNIX every device is actually treated as a file. We will look at this concept in more detail in the next unit, but here let us see the behaviour of a special device file in UNIX, called the null device. This is available in the device directory just as other devices are. Anything written out to the null device simply disappears, and it is thus an infinite sink. Trying to read from the null device always results in an end of file. So if you want to just discard the standard output, you just redirect it to the null device.

```
$ ls -l  /dev/null 2 errfile
```

This will throw away the output of the command but will save the error messages in errfile. Likewise, to discard the error messages but save the output, you could say

```
$ ls -l  dirfile 2 /dev/null
```

You now know how to suppress all output from a command, but why would one want to do that? Possibly if the command did perform some operations on a device or disk file and you want it to work silently without showing anything on the terminal.

```
$ mv file1 file2  /dev/null 2&1
```

will do the move without any terminal output even if there are error messages from it. This is not a good example though, for the command does not write to the standard output.

2.8 ADVANCED FEATURES

Let us now look at a few capabilities of the shell programming language which you could consider somewhat advanced. First we will learn to do arithmetic. This can be done with the expr command

```
$ expr 2 + 3
5
```

Every number and operator is a separate argument to the expr command. So all of them have to be separated by spaces. If you do not do so, you will get an error. The command works only on positive or negative integers. You can use the operators +, -, * for multiplication, / for division and % for the remainder or modulo operation. Let us write a shell script to find the average of the numbers entered.

```
$ cat average
:
total=0
count=$#
for i
do
total=`expr $total + $i`
done
avg1=`expr $total / $count`
avg2=`expr $total % $count`
avg2=`expr $avg2 \* 100 / $count`
echo "The average is $avg1.$avg2"
```

We initialise the variables total and count and compute the total by adding the value of each argument until there are no more arguments. Since we want a somewhat more accurate average than to the nearest integer, we get the result to two decimal places in a roundabout fashion. We first find the integer part in avg1. To find the decimal part, we find the remainder after the division, multiply it by 100 and divide it by the number of entries. This gives us the digits after the decimal point. We then print out the answer by printing out both the variables separated by a decimal point.

Note that the * operator has to be escaped with a backslash \, otherwise it would have the special meaning of all the files in the directory to the shell. The expr command has some more capabilities which we will not discuss here.

Let us now look at another feature of the shell, that of trapping interrupts and taking some action depending on them. We will illustrate this with a simple example. Suppose you have written a shell script which interchanges the names of the two files given to it as arguments. That is, if there are two files called chicken and egg, then

```
$ interchange chicken egg
```

renames chicken to egg and egg to chicken. As a programmer you will realise that to do this

you need to use a temporary file. So a bare bones script (without any precautions or error checking) to do the job will look like this

```
$ cat interchange
mv chicken tmp
mv egg chicken
mv tmp egg
```

There is a fair amount of checking this script needs to do in order to be useful. The files must both exist and there should not already be a file called tmp. Assuming that these things are taken care of, what happens if the user breaks the script in between? Depending on the actual instant when this is done, you will have a different situation, but certainly the effect you wished for will not have occurred, for one of the files will be named wrongly and you will have a tmp file lying around. What you want in this case is a way to prevent the user from interrupting the script until it is complete.

In other, more realistic situations one uses temporary files to hold temporary results. These files are deleted in the script after they are no longer needed. If your script is interrupted you would not want these files to be lying around.

The shell has a command called trap which allows you to achieve what you want. If you want to prevent your script from getting interrupted by the break key, you can say

```
trap '' 2
```

The quotes are the first argument to the trap command, which holds the command or commands (separated by ;) to be executed whenever the script receives any of the signals specified by the signal numbers in the subsequent arguments. You can look at the signal numbers and their meanings for your system by examining the file "/usr/include/signal.h". A signal gets sent to your process will result in the action you asked for. Here since there is no command specified, the effect will be to do nothing when the shell script receives the signal number 2, which is sent by the break key. To remove a temporary file you create, you can say instead

```
trap 'rm -f tmp$0$$ 2/dev/null; exit' 2 15
```

$$ stands for the current process number and tmp$0$$ is a reasonable way of almost ensuring that the name of the temporary file you create will not conflict with any other existing filename. Signal number 15 is the default kill signal sent to a process when you kill it. However note that signal number 9 is the kill signal which cannot be caught or ignored, and so it is no use giving signal number 9 as one of the signals in your list.

In the C-shell interrupts are handled differently and you should consult the documentation for details.

One more useful feature is the ability to give the input to a command which actually takes its input from the standard input normally. You already saw how you can ensure that if any command in the shell script needs to take some input from the standard input, you can supply that input from the keyboard just as you would have done if the command had been run from the terminal. But that is not always what you want or need. Sometimes what is more important is to pass fixed input to a shell command. For example you might want to write a shell script which can be called with a filename argument, and which locates and deletes all lines containing the word specified as the second argument. You already know how to do this in ed interactively. But how do you give this input in the script itself?

To illustrate this with a simple example, let us take up the case where you want to give a message if a person gives the wrong number of arguments. You could use echo, but let us do this with cat instead.

```
if test $# -ne 4
then
cat < this
What a stupid mistake
this
fi
```

The feature shown here is called a here document and is introduced by the notation <. The word immediately after the < is a label, and everything upto the first occurrence of that label first thing on a line by itself is taken to be input to the cat command. So you can embed input for your command in the shell script itself. It should be clear that you cannot use this facility with commands that do not read from the standard input.

You should now be able to write a shell script that does the things we had talked of in the introduction to here documents. The only part which needs to be done carefully is the input to the ed command.

Before we end this unit, let us talk briefly about how to debug shell procedures. Although as experienced programmers you will not find it difficult to figure out where the error lies by using the echo command judiciously, you could find the facilities of the shell useful. For example you can run a file with the -v option for verbose output, wherein each command is printed and then executed. This is like having an echo command after every line.

$ sh -v Concat

Another option is -x, where every command line is shown with a + sign at the beginning, and the values of variables before and after the command are printed. To be safe you can decide to use the -n (noexecute) option, where the commands are read but not executed.

2.9 SUMMARY

You have learnt to program in the shell's programming language. We have kept the scripts introduced here very simple, because of lack of space and the necessity of introducing the concepts rather than expounding on complex features.

You should use shell scripts whenever you find yourself doing the same operation again and again. Shell scripts can be used as prototyping tools before you write something in a programming language, because shell scripts can be written quickly although they are slow to execute.

Look at the system files. Many of them are shell scripts and some of them are quite sophisticated, using tricks and artifices which we were not able to present here.

2.10 MODEL ANSWERS

Check Your Progress 1

1. This will not list filenames which start with a period (.). To get these as well, one needs to say

   % ls * .*

2. This is because the length of the arguments to ls becomes too much in the second case. The total number of characters in a command, including all its arguments, has an upper limit which gets exceeded.

3. One can say simply

   % echo * .*

Check Your Progress 2

1. If the file is fixedfile, then Cat should contain

   % cat Cat

   tr '[a-z]' '[A-Z]' fixedfile

   set tabstop=4

   set terse

2. No model answer.