Unix Shell Programming

1 Introduction

The UNIX shell provides a command line interface to the UNIX operating system. In addition to providing a simple command line facility the shell is powerful programming language. This programmability makes the shell a much more powerful way of interfacing with the system than using a graphical interface. In this chapter we will systematically examine the shell as a programming language.

2 Command Structure

The structure of a shell command is simple. It consists of a series of *words* separated by white space. For example, the `echo` prints any arguments to its standard output stream, separated by spaces.

```
$ echo hi there
hi there
```

Commands are usually terminated by a newline, but a semicolon `;` is also a command terminator. Thus typing the line

```
echo the time is ; date
lots of lines
and more
```

to the shell, will cause the two commands to be run, one after the other.

```
$ echo the time is ; date
the time is
Wed Mar 1 10:05:00 NZDT 1995
```

If we try sending the output from the line above through a pipe to the `wc` command (forming a *pipeline* command), the result is as follows:

```
$ echo the time is ; date | wc
the time is
      1       6      30
```

It is only the output of the second command which is filtered by `wc`. This is because connecting two commands with a pipe forms a single command. Thus `date | wc` is regarded as a single command which is separated from the command `echo the time is` by `;`.

If it really desired to pipe the output of both commands through `wc`, they can be grouped with parentheses.

```
$ (echo the time is ; date) | wc
       2       9      41
```

Commands can also be terminated by the ampersand character `&`. This works exactly like `;` or a newline, but it tells the shell not to wait for the command to complete before prompting for a new command. Typically this is used to run long running commands "in the background" while you continue to type interactive commands. (Now that UNIX has a multi-window graphical interface this feature is used much less than it used to be. However, you will still need to "background" jobs if you want them to continue running after you log out.)

```
$ long-running-command &
1525
$
```

The shell prints the process-id number of the command and prompts immediately for another command. The process-id is a unique value which identifies the command for as long as it runs.

The `&` operator terminates commands and because a pipelines are commands, they can be terminated by `&`. This means that we could send the output of a long-running command directly to the printer using the `lpr` command as follows:

```
$ long-running-command | lpr &
```

Most commands accept *arguments* on the command line. These arguments may be the names of files, or a pattern to search for, or an option flag (an argument beginning with `-`). The various special characters interpreted by the shell such as `>`, `<`, `|`, `;` and `&` are *not* arguments to the commands the shell runs. They instead control how the shell runs them. For example, the command

```
$ echo hello > junk
```

tells the shell to redirect the output of the command `echo` into the file `junk`. Neither `>` nor `junk` are arguments to the command `echo`; they are interpreted by the shell and never seen by `echo`. In fact, the command

```
$ > junk echo hello
```

is identical (but less intuitive).

3 Metacharacters

The shell recognises a number of other characters as special. The most commonly used is the asterisk `*` which can be used to match filenames. For example, the following command echos the name of all files in the current directory which start with j.

```
$ echo j*
junk
```

Characters like `*` which have special properties are known as *metacharacters*. There are a lot of them. The following table gives the complete list.

>	cmd > file, direct standard output to file
>>	cmd >> file, append standard output to file
<	cmd < file, take standard output from file
|	cmd1 | cmd2, connect standard output of cmd1 to standard input of cmd2
<< str	*here document*: standard input follows, up to next str on a line by itself
*	match any string of zero or more characters in filenames
?	match any single character in filenames
[ccc]	match any character from ccc in filenames ranges like 0-9 or a-z are legal
;	command terminator: cmd1 ; cmd2 does cmd1 then cmd2
&	like ; but doesn't wait for cmd1 to finish
`` `...` ``	run command(s) in ...; output replaces `` `...` ``
(...)	run command(s) in ... in a sub-shell
{...}	run command(s) in ... in current shell
$1, $2 etc.	$0 ··· $9 replaced by arguments to shell script
$*var*	value of shell variable *var*
${*var*}	the value of *var*; avoids confusion when concatenated with text
\c	take character c literally, \newline discarded
'...'	take ... literally
"..."	take ... literally after $, `` `...` `` and \ interpreted
#	if # starts word, rest of line is a comment
var=value	assign to variable var
cmd1 && cmd2	run cmd1; if successful run cmd2
cmd1 || cmd2	run cmd1; if unsuccessful run cmd2

Given the number of shell metacharacters, there needs to be some way to tell the shell "leave it alone". The simplest way to protect characters from being interpreted is to enclose them in single quote characters. For obvious reasons this is known as *quoting*.

```
$ echo '***'
***
```

It is also possible to use double quotes "...", but the shell will process any $, `` `...` `` and \ it finds in "...", so don't use this form of quoting unless you want this kind of metacharacter expansion to take place.

Another possibility is to put a backslash `\` in front of *each* character you want to protect from the shell, as in

```
$ echo \*\*\*
```

Quotes of one kind will protect quotes of another.

```
$ echo "Isn't this fun?"
Isn't this fun
```

and they don't need to surround the whole argument.

```
$ echo A'* ?'
A* ?
```

Note that in this last case, there is only one argument to `echo`. The space between `*` and `?` is part of the argument because its special function as a delimiter was removed by the quoting.

Quoted strings can contain newlines:

```
$ echo 'hi
> there'
hi
there
$
```

The string "`>`" is the *secondary prompt* printed by the shell when it expects more input to complete a command.

In all these examples, the quoting of metacharacters prevents the shell from trying to interpret them. The command

```
$ echo j*
```

echos the all filenames beginning with `j`. The command `echo` knows nothing about files or shell metacharacters; the interpretation of `*` and any other metacharacters is supplied by the shell. (This is in contrast to DOS where it is the individual programs which are responsible for handling metacharacter expansion (making programming much harder).)

A backslash at the end of a line causes the line to continued; this is the way to type a very long line to the shell.

```
$ echo abc\
> def\
> ghi
abcdefg
$
```

Notice that the newline is discarded when preceded by a backslash, but retained if is included in quotes.

The metacharacter `#` is used for shell comments. If a shell word begins with `#`, it and the rest of the line are ignored.

```
$ echo hi # there
hi
$ echo hi#there
hi#there
```

4 Creating New Commands

Given a sequence of commands that is to be repeated more than just a few times, it would be convenient to make the sequence into a single new command. To be concrete, suppose that intent to count users frequently with the pipeline

```
$ who | wc -l
```

The command `who` prints the names of logged in users (and some other details) one per line. This is piped to `wc -l` which counts the number of lines printed. (Under X Windows this is a poor way to count users because each xterm window is counted as a login. We will fix this later.)

The first step in creating a new command is to create a file which contains "`who | wc -l`". This can either be done with a text editor, or more creatively

```
$ echo 'who | wc -l' > nu
```

(Without the quotes, what would appear in `nu`?)

The shell is just a program; its name is `sh`. It can be invoked the same way as any command in the system. In particular, we can make the shell read its input from a file by using input redirection.

```
$ sh < nu
       4
```

Here the shell has read the command from the file `nu` and executed it.

Like other programs, the shell can take arguments. It interprets its first argument as a file to use as input. Thus

```
$ sh nu
```

will produce the same result. (Note that the command `sh < nu` is not the same as `sh nu`. The standard input to commands in the first command is taken from the file `nu` whereas the input to those in the second command is taken from the terminal.)

While `sh nu` is a shorthand way of determining the number of users, it still does not look like a regular command. In UNIX if a file is executable and contains text it is assumed to contain shell commands. When the name of such a file is typed, a new shell is run to interpret the commands in the file. Such a file is called a *shell script*. The file `nu` can be made executable with the command:

```
$ chmod +x nu
```

Thereafter it can be invoked simply by typing its name.

```
$ nu
       4
```

To complete the process of making `nu` appear like a ordinary UNIX command, it must be moved to one of the standard places where the shell looks for commands. The usual system places are inaccessible to ordinary users, but each user can create a "bin" directory in their home directory for their own commands. This directory can be created and the command `nu` installed in it as follows.

```
$ mkdir bin
$ mv nu bin
```

The command mkdir creates a new directory and the command mv moves nu into it. If you created nu somewhere other than your home directory the same effect can be obtained by

```
$ mkdir ~/bin
$ mv nu ~/bin
```

The ~ is shorthand for your home directory. (You can refer to other peoples home directories in a similar fashion; ~fred is the home directory of the user "fred".)

5 Command Arguments and Parameters

Although nu is adequate as it stands, most shell programs interpret arguments, so that for example, filenames and options can be specified when the program is run.

Suppose we want to make a program called cx to change the mode of a file to executable, so

```
$ cx nu
```

is shorthand for

```
$ chmod +x nu
```

We already know how to do most of this. We need a file called cx whose contents are

```
chmod +x filename
```

where filename is to the value passed as the first argument to the shell.

When the shell executes a shell-script, each occurence of $1 in the file is replaced by the first argument, each $2 is replaced by the second argument and so on through $9. If the file cx contains

```
chmod +x $1
```

it will behave as we want. When the command

```
$ cx nu
```

is run, the sub-shell replaces "$1" by its first argument "nu", and it is nu which is made executable.

Suppose we want to handle more than one argument. It is clear that we could try something like

```
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```

This does indeed work for up to 9 arguments. If fewer arguments are supplied, the remaining parameters have null strings substituted and the command is executed with fewer parameters.

This technique does not work for more than 9 parameters because a parameter like $10 is parsed as $1 followed by a zero. Anticipating this problem the shell provides the shorthand $* which means "all the arguments". A more general way of writing cx is thus

```
chmod +x $*
```

which works regardless of how many arguments are passed.

In all the commands we've written so far the arguments have always been filenames. This need not be the case. We will now consider a program which looks up phone numbers in a small data base. We'll assume that this data base is kept in the home directory and is called *Phonebook.* The lines of this file are of the form

```
University      373-7599
Statistics      x87510
Fax             373-7018 or x87018
Ronald Fisher   x81234
Jerzy Neyman    x81235
Karl Pearson    x81236
```

We are interested in writing a command called `phone` which could be used as follows:

```
$ phone pearson
Karl Pearson    x81236
$ phone ron
Ronald Fisher   x81234
```

The argument to `phone` is a pattern which is to be searched for in the file `Phonebook`.

The hard part of writing a program like this is the pattern matching. Fortunately, that part is already provided by the command `grep`. (`Grep` is a contraction of Get Regular Expression and Print.) The command

`grep` *pattern file* ...

searches for the given *pattern* in the specified *file* and prints out any lines which contain it. We will also make use of the `-i` option to `grep` which causes character comparisons to ignore case.

We can create the `phone` shell script with the following contents:

```
grep -i $1 Phonebook
```

When invoked as in the examples above the command works well, but it has a problem. If I search for the pattern "`karl pearson`" an error occurs.

```
$ phone 'karl pearson'
grep: pearson: No such file or directory
Phonebook:Karl Pearson    x81236
```

The problem is that the pattern is substituted in place of `$1` in the script, yielding the command:

```
grep karl pearson Phonebook
```

This will search for the pattern `karl` in the files `pearson` and `Phonebook`. The first of these files doesn't exist.

The problem is that we need to prevent the space between "`karl`" and "`pearson`" being interpreted as an argument delimiter. This can be done using double quotes. This kind of quoting will preserve the space as part of the argument, but will allow the expansion of `$1`. When the value of `$1` is substituted into the command

```
grep "$1" Phonebook
```

the command will be expanded to

```
grep "karl pearson" Phonebook
```

and this will do what is desired. The command could also be written

```
grep "$*" Phonebook
```

which would allow us to interrogate the phone data base as follows:

```
$ phone karl pearson
Karl Pearson    x81236
$ phone karl
Karl Pearson    x81236
```

(If there are multiple spaces between the given and family names in `Phonebook` the pattern match will fail).

6 Program Output as Arguments

Filename expansion from metacharacters is the most common way to generate arguments (apart from providing them explicitly). There is another of generating arguments which is fairly commonly. This is to use the output of a program as arguments to another. The output of a command can be placed on the command line by enclosing it in back quotes `‘...‘`.

```
$ echo At the tone the time will be ‘date‘.
At the tone the time will be Wed Mar  1 10:15:32 NZDT 1995.
```

Back quotes are interpreted inside double quotes.

```
$ echo "At the tone
> the time will be ‘date‘."
At the tone
the time will be Wed Mar  1 10:15:45 NZDT 1995.
```

This may not seem very useful, but here is an example which shows how the feature can be used.

As noted earlier, the command `grep` can be used to find patterns in a file. When invoked as

```
$ grep -l pattern file ...
```

`grep` will print the names of those files in the list `file ...` which contain the specified pattern. Thus if we wanted to edit all the files in a directory containing the word “money”, we could issue the command:

```
$ vi ‘grep -l money *‘
```

A particular use of backquotes is to reset the values of the command line arguments. This is done with the shell `set` command. The command

```
$ set word1 word2 ...
```

resets the value of `$1` to *word1*, `$2` to *word2*, etc. The combination of backquotes and the `set` command can be use to make the output from a command available in `$1`, `$2`, etc. For example the command

```
$ set `date`
```

sets `$1`, `$2`, etc. to the words that make up the output from `date`. Thus `$1` is the day of the week, `$2` is the month, `$3` is the day of the month, `$4` is the time, `$5` is the time zone and `$6` is the year.

An additional use for backquotes is the manipulation of shell variables. We now turn to these.

7 Shell Variables

Like most programming languages, the shell has *variables*. We have already encountered the variables `$1`, `$2`, etc. which are variables which contain the arguments typed to a shell program.

Shell variables are created by assigning them a value. For example, the assignment

```
$ x=fred
```

creates a variable `x` (if it doesn't already exist) and assigns it the value "`fred`". To access a shell variable, its name is prefixed by `$`.

```
$ echo $x
fred
```

Some variables are special to the shell and you should be careful not to reset them by accident. The variable `PATH` holds the shell's *search path* — the list of directories where the shell looks for commands. Resetting `PATH` can cause the shell to lose track of where to find commands.

```
$ PATH=fred
$ ls
sh: ls: command not found
$ who
sh: who: command not found
```

No permanent damage is done by accidentally destroying a variable like this since it could be reset by assignment or, since each window open on the display has its own shell with its own set of shell variables, you can always close the window you are working in and open another. The following table contains a list of some of the important special shell variables. The manual entry for `sh` (or `bash`) describes others.

`$#`	the number of arguments
`$*`	all arguments to the shell
`$@`	similar to `$*` except when quoted with `"`
`$-`	options supplied to the shell
`$?`	value returned by the last command executed
`$$`	the process-id of the shell
`$!`	the process-id of the last command started with `&`
`$HOME`	the user's home directory; default argument to `cd`
`$IFS`	the list of characters which separate commands in arguments
`$MAIL`	file which, when changed, triggers "you have mail" message
`$PATH`	list of directories to search for commands (`:` separated)
`$PS1`	primary shell prompt, default `$`
`$PS2`	secondary shell prompt (continued commands), default `>`

A shell's variables and their values are made available to sub-shells, however, assignments made in the sub-shell are not reflected in the top-level shell. The following example demonstrates this.

```
$ x=one
$ echo $x
one
$ (echo $x ; x=two ; echo $x)
one
two
$ echo $x
one
```

The first assignment takes place in the top-level shell and the second in the sub-shell. The first `echo` refers to the top-level `x`, the second and third to the `x` in the sub-shell and the final one to the top-level `x` again.

Although shell variables are propagated to sub-shells, they are not usually propagated to shell scripts which are run from the top-level shell. The example below uses a shell script called `xecho` which simply echos the value of the variable `x`.

```
$ cat xecho
echo $x
$ x=yes
$ xecho

$ echo $x
yes
```

The variable `x` is set in the top-level shell, but not the shell which executes the `xecho` script.

It is possible to make the value of a variable available to shell scripts. This is done by marking the variable for *export* in the top-level shell.

```
$ x=yes
$ export x
$ xecho
yes
```

The collection of all exported variables and their values form the *environment* provided by the shell to the shell scripts it runs (and also to compiled programs).

8 Input-Output Redirection

We saw in chapter two that the standard input to and standard output from commands can be redirected with the redirection symbols `<`, `>` and `>>`. We saw also that the output of one command can be used as input to another if the commands are connected with a pipe (denoted by the pipe symbol `|`).

In addition to standard input and output, commands also have a standard error stream to which they can write. By default the standard error is connected to the user's terminal. It is not redirected by the symbols `>` and `>>`. The standard error was invented so that error messages would always appear on the terminal.

```
$ cat xxx yyy > zzz
cat: xxx: No such file or directory
cat: yyy: No such file or directory
```

If the error messages were written to standard output, they would be sent to the file `zzz` and the user would be blissfully unaware that there was a problem.

The three default streams are numbered by the integers `0`, `1` and `2` corresponding to standard input, standard output and standard error. These integers are sometimes used to explicitly redirect the output streams. For example, the command

```
$ cat xxx yyy > zzz 2> errors
```

redirects the standard output of the `cat` command into `zzz` and simulaneously redirects the standard error into the file `errors`.

The following table shows the ways in which the shell's input and output streams can be manipulated.

`>  file`	direct standard output to `file`
`>> file`	append standard output to `file`
`<  file`	take standard input from `file`
`cmd1 | cmd2`	use standard output from `cmd1` as standard input to `cmd2`
`^`	obsolete synonym for `|`
`2>  file`	direct standard error to `file`
`2>> file`	append standard error to `file`
`2>&1`	print standard error messages to standard output
`1>&2`	print standard output messages to standard error
`<< string`	here document: take standard input until next `string` at the beginning of a line; substitute for `$`, `` `...` `` and `\`
`<< \string`	here document with no substitution
`<< 'string'`	here document with no substitution

There are two idioms in the table which we have not discussed but which are relatively important.

The notation `2>&1` tells the shell to write standard error output to the standard output stream. It is also possible to redirect standard output to the standard error stream. This is most commonly used as follows to produce error messages on the standard error stream.

```
echo ... 1>&2
```

The shell provides a mechanism for including the standard input to a command in the same file as the command rather than in a separate file. This means that a shell script can be completely self-contained. Our phone number program could be written using this mechanism as follows.

```
grep -i "$*" << END
University      373-7599
Statistics      x87510
Fax             373-7018 x87018
Ronald Fisher   x81234
Jerzy Neyman    x81235
Karl Pearson    x81236
END
```

The `<<` signals the start of the construction; the word which follows (`END` in our example) is given on a line by itself to signal the end of the input. The standard jargon for input specified in this way is a *here document.* The shell substitutes for `$`, `` `...` ``, and `\` in here documents, unless some part of the delimiting word after `\dl` is quoted with quotes or a backslash. In the `phone` script above we could turn off metacharacter substitution by changing the first line of the script to

```
grep -i "$*" << 'END'
```

9 For Loops

Programming languages become useful when it possible to direct the flow of computation. The shell has a number of constructions which provide this kind of *control flow*. The most commonly used of these is `for`. It is the only control-flow mechanism which is commonly typed interactively. The syntax of the `for` statement is:

```
for var in list-of-words
do
    commands
done
```

For example, a command which will echo file names on two a line is as follows.

```
$ for i in *
> do
> echo $i
> done
```

The `i` can be any shell variable. Note that the value of the variable is accessed with `$i`, but the `for` loop refers to `i`.

Using looping can be overdone. For example, the command

```
$ for i in $*
> do
>     chmod +x $i
> done
```

is much less efficient than

```
$ chmod +x $*
```

because it causes multiple `chmod` commands to be run.

We will now present an example which shows how the various shell features which we have listed in this chapter can be used to create a very useful command. The command is called `shar` which stands for shell archiver. Given a list of file names, the command creates single file archive which contains all the files in a form which allows them to be extracted in a simple way. For example, the command

```
$ shar *.h *.c > cfiles.shar
```

can be used to bundle a collection of C source files into an archive. This archive can then be distributed by by e-mail. When the file is received, its contents can be extracted with the command

```
$ sh cfiles.shar
```

The basic trick used here is to package files up as “here documents” which, in conjuction with the `cat` command, can be restored to separate files. The basic form of the command is a s follows.

```
# shar - simple version
# group files into a shell archive

echo '# to unbundle, sh this file'
for i in $*
do
    echo "echo $i 1>&2"
    echo "cat >$i << 'End-of-$i'"
    cat $i
    echo "End-of-$i"
done
```

To see how this works let's apply shar to a file called `A` which contains `AAAA`

```
$ shar A > files.shar
```

The file `files.shar` contains

```
# to unbundle, sh this file
echo A 1>&2
cat >A << 'End-of-A'
AAAA
End-of-A
```

This is a shell script which will recreate the file `A` when processed by `sh`. Before the file is extracted from the archive, its name is echoed to the standard error.

This is a very nice program – a few lines of code which do something simple, useful and elegant. Of course, it is far from perfect. In particular, a line of the form `End-of-A` in the file `A` would spell disaster as would specifying a directory as an argument to `shar`. These difficulties can be overcome, but to write a bulletproof version we will need some additional shell features.

10 Case Statements

The shell has statements which allow conditional branching within command scripts (or interactively). The shell `case` statement allows branching to one of several alternatives based on the value of a word. The syntax of the `case` statement is as follows:

```
case word in
    pattern1) statements ;;
    pattern2) statements ;;
    ...
esac
```

The `case` statement compares *word* to each of the *pattern*s, top to bottom, and performs the *commands* associated with the first pattern to match *word*. The allowable patterns are a slightly generalised version of the shell's pattern matching for filenames. The rules are given in the following table.

*	match any string, including the null string
?	match any single character
[ccc]	match any of the characters in ccc [f-h0-2] is equivalent to [fgh012]
"..."	match ... exactly; quotes protect special characters. Also '...'
\c	match c literally
a | b	in case expressions only, matches either a or b
/	in filenames, matched only by an explicit / in the expression; in case, matched like any other character
.	as the first character of a filename, is matched only by an explicit . in the expression

A simple example of a case statement is shown in the following fragment which shows how the number of arguments to a shell script can be checked.

```
case $# in
0) echo error: no arguments 1>&2 ;;
1) commands for one argument ;;
2) commands for two arguments ;;
*) echo error: too many arguments 1>&2 ;;
esac
```

(Notice the redirection of the output of the echo commands to standard error.)

This can be used to redesign the interface to the cal command. Recall that given one argument, cal interprets that argument as a year, and given two, it interprets them as a month and year. Users often type a command like "cal 7" expecting to see a calendar for July of the current year rather than a calendar for the year 7. Alternatively, a user might type "cal july 1995" only to be informed that july is not a valid month.

The following script acts as a font end to the system cal which corrects these problems.

```
# A friendly interface to cal

case $# in
0) set ‘date‘ ; m=$2 ; y=$6 ;;
1) m=$1 ; set ‘date‘ ; y=$6 ;;
2) m=$1 ; y=$2 ;;
*) echo error: incorrect number of arguments to $0 1>&2
   exit 1 ;;
esac

case $m in
jan*|Jan*) m=1 ;;
feb*|Feb*) m=2 ;;
mar*|Mar*) m=3 ;;
apr*|Apr*) m=4 ;;
may*|May*) m=5 ;;
jun*|Jun*) m=6 ;;
jul*|Jul*) m=7 ;;
```

```
aug*|Aug*) m=8 ;;
sep*|Sep*) m=9 ;;
oct*|Oct*) m=10 ;;
nov*|Nov*) m=11 ;;
dec*|Dec*) m=12 ;;
[1-9]|10|11|12) ;;
*) y=$m ; m="" ;;
esac

/usr/bin/cal $m $y
```

The first `case` statement checks the number of arguments and uses the `date` command to obtain either the current month and year, or the current year if these were not specified on the command line. An incorrect number of arguments results in an error message printed to the standard error stream. Note the use of `$0` which is name by which the shell script was invoked.

The second `case` statement converts alphabetically specified months into a number which can be passed to the system `cal`. Anything unrecognised by this `case` statement is taken as a numerical year. If this is not the case, the system `cal` will produce an error message when invoked.

There is a minor problem with the script. It is not possible to obtain calendars for the first 12 years of the early Christian era. This is a very minor problem because the calendars produced by `cal` were not in effect during that period anyway.

11 If Statements

In addition to the branching provided by the `case` statement, the shell also has an `if` statment. The shell `if` statement has one of the forms

```
if cmd
then
    commands-if-true
fi
```

or

```
if cmd
then
    commands-if-true
else
    commands-if-false
fi
```

UNIX commands all return a *status* on their completion. If a command is successful it returns a value of zero and if it fails, the returned value is non-zero. The `if` statement examines the value returned by the command *cmd.* If the command was successful (i.e. it returned zero) then the *commands if true* commands are executed, otherwise the *commands if false* commands are executed.

Most commonly, the command tested by `if` statements is the system command `test`. The `test` command serves a variety of functions. A full selection of the possible functions carried out by the `test` command is detailed in the manual entry for `test`, but we will list some of the most common ones.

```
test -f filename
```

tests to see if there is a (regular) file called `filename`.

```
test -d dirname
```

tests to see if there is a directory called `dirname`.

```
test -x name
```

tests to see if there is a file (or directory) called `name` which has executable mode.

Tests like these can be combined with the logical conjunctions `-o` (or) and `-a` (and). Thus, the command

```
test -f name -a -x name
```

tests to see if there is an ordinary file called `name` which is executable.

We can examine how these commands work in the following dialog.

```
$ test -d /usr/bin
$ echo $?
0
$ test -f /usr/bin
$ echo $?
1
```

The first test shows that `/usr/bin` is a directory because zero was returned (recall that `$?` is set to the value returned by the last command executed) and the second test shows that `/usr/bin` is not an ordinary file because a non-zero value was returned.

Using the facilities developed so far, we will now create a command (called `which`) which determines the location of specified command. Recall that the shell variable `PATH` contains a colon separated list of the locations which are searched for commands. The dialog

```
$ echo $PATH
:/usr/local/bin:/usr/ucb:/usr/bin:/bin
```

indicates that commands will be searched for in the locations

```
.  # the current directory
/usr/local/bin
/usr/ucb
/usr/bin
/bin
```

in that order (an empty specification in the list is taken to be the current directory "`.`").

The basic command will have the form:

```
for each directory in the PATH
  if the command is in the directory
    echo the full pathname of the command and exit
```

The first difficulty is to turn the colon separated `PATH` variable into a blank separated list of words which can be processed by the `for` loop. This can be done with the stream editor `sed` as shown in the following loop which echos each directory in the path.

```
for dir in `echo $PATH | sed 's/^:/.:/
                              s/::/:.:/g
                              s/:$/:./
                              s/:/ /g'`
do
    echo $dir
done
```

The `sed` command first replaces empty entries in `PATH` by the current directory and then substitutes blanks for all colons which appear in `PATH`. Backquotes then substitute this list of words into the `for` command.

The rest of the command is easy. We simply test to see whether or not the specified command is in each of locations in the path. The full command appears below

```
# which: determine where a command is located

case $# in
1) ;;
*) echo "usage: which command" 1>&2 ; exit 2 ;;
esac

for dir in `echo $PATH | sed 's/^:/.:/
                              s/::/:.:/g
                              s/:$/:./
                              s/:/ /g'`
do
    if test -f $dir/$1 -a -x $dir/$1
    then
        echo $dir/$1
        exit 0;
    fi
done

exit 1
```

If the command is found, its name is printed on the standard output and 0 is returned (by the `exit` statement) to indicate success. If the command is not found, nothing is printed and 1 is returned to indicate failure.

The command can be used as follows:

```
$ which
usage: which command
```

```
$ which which
./which
$ which cat
/bin/cat
```

There is one potential problem with this command. A user might redefine `echo` and/or `test` in a way that might interfere with its operation. (Most modern UNIX systems use builtin versions of `echo` and `test` so that this is not a problem. On older UNIX systems this kind of precaution is important.) A simple solution to this problem is to switch the value of `PATH` to something which does not contain user-defined commands which might interfere with the script's operation. This would mean changing the `for` loop in the script as follows:

```
userpath=$PATH
PATH=/usr/bin:/bin
for dir in `echo $userpath | sed 's/^:/.:/
                                  s/::/:.:/g
                                  s/:$/:./
                                  s/:/ /g'`
```

With this change, the shell will only use commands which are in the system directories, so that user defined commands cannot interfere.

12 While and Until Loops

In addition to `for` loops, the shell has two other looping statements. These are `while` and `until` loops. The `while` loop has the form

```
while command
do
    commands-to-be-executed-while-command-remains-true
done
```

and the `until` loop the form

```
until command
do
    commands-to-be-executed-until-the-command-is-true
done
```

The conditional command which controls these loops can be any command.

These statements are useful for monitoring of ongoing processes on the computer. Here is a simple example which watches for someone to log on.

```
while sleep 60
do
    who | grep scott
done
```

The command repeatedly sleeps for 60 seconds and then examines the list of people logged in to the machine for someone called `scott`. When `scott` is found to be logged on, a line is printed.

The command could alternatively be written:

```
until who | grep scott
do
    sleep 60
done
```

In fact this second form is superior in two ways. The first command will always pause for 60 seconds before printing anything while the second command will indicate immediately that `scott` is logged in. Additionally, the first command must be terminated manually, whereas the second will terminate as soon as `scott` logs on.

We can package this statement up in shell script which we'll call `watchfor`. As before, we'll do some argument checking and make sure that the command cannot be disrupted by a user who redefines the commands which are used by the script. Rather than have a fixed name built into the command we'll allow the user to specify the name on the command line. We'll also change the pattern matching command from `grep` to egrep. This change means that it will be posible to watch for several peoples logging on.

```
# watchfor - watch for someone to log in

PATH=/usr/bin:/bin

case $# in
1) ;;
*) echo usage: watchfor person 1>&2; exit 1 ;;
esac

until who | egrep "$1"
do
    sleep 60
done
```

The command could be invoked as

```
$ watchfor scott
```

to watch for a single person logging on, or as

```
$ watchfor 'scott|lee|triggs'
```

to watch for one of several people logging in (patterns of the form *pattern* | *pattern* are allowed by `egrep` and are interpreted as either the first or second pattern — or both).

As a more complicated example, we'll write a script which watches add logons and logoffs, reporting as people come and go. The basic command is as follows

```
# watchwho - monitor logins and logoffs

PATH=/bin:/usr/bin
new=/tmp/ww1.$$
old=/tmp/ww2.$$
who > $old
```

```
while true
do
        who > $new
        diff $old $new && echo
        mv $new $old
        sleep 15
done | egrep '<|>'
```

The command works by keeping a list of the people who are currenly logged into the system. Every 15 seconds, a new list is generated and compared with the old list using the command `diff` which produces a description of the differences between files. Lines in this output which contain `<` or `>` are selected by the `egrep` command.

The reason for making this selection is that the output from diff contains the differing lines from the files marked with `<` if they are from the first file and with `>` if they are from the second file. A line will be present in the first file and not in the second when someone logs off and a line will be present in the second file and not the first when someone logs on.

Thus when someone logs on, a line of the form

```
> user    additional information
```

will be printed and, when someone logs off a line of the form

```
< user    additional information
```

will be printed.

The output produced by the script does indicate who logs on and who logs off, but it is not as simple to understand as it might be. It would be much easier to understand if the `>`s were replaced by `On:` and the `<`s were replaced by `Off:`

This seems like a job tailor made for `sed`. The command

```
sed -n 's/>/On: /p
        s/</Off:/p'
```

prints only those lines which contain either `<` or `>`, and it substitutes `Off:` and `On:` for those symbols. There is however a problem with using `sed` to do the job. For technical reasons `sed` does not print an output line until it has read its next input line. Thus we can't just replace the `egrep` command with the `sed` command above because we will always be *one line behind.* Fortunately there is a simple solution, we force a blank line to be printed after the `diff` command by using `echo` with no arguments. This forces the line we want printed out of `sed`

Finally, the script as we have written it creates two files in `/tmp` which it does not remove. We can force the script to remove these files when it terminates with the line

```
trap 'rm -f $new $old; exit 1' 1 2 15
```

This says that if the shell receives a signal numbered 1, 2, or 15, it should execute the command in quotes.

The signals that a shell command can receive are as follows.

0	shell exit (for any reason, including end of file)
1	hangup (lost modem connection etc.)
2	interrupt i.e. `^C`
3	quit (`^\`; causes the program to dump core)
9	kill (cannot be caught or ignored)
15	terminate, default signal generated by the `kill` command

The `trap` statement is really saying that if the script is interrupted it should remove its temporary files. The particular details of the signal handling are somewhat beyond the level of material presented here. It is really sufficient to understand that the `trap` statement above will remove the temporary files.

With these changes we now have a complete command for watching users logon and logoff.

```
# watchwho - monitor logins and logoffs

trap 'rm -f $new $old; exit 1' 1 2 15

PATH=/bin:/usr/bin
new=/tmp/ww1.$$
old=/tmp/ww2.$$
who > $old

echo Initial list of users:
echo
cat $old
echo

while true
do
    who > $new
    diff $old $new
    echo
    mv $new $old
    sleep 2
done | sed -n 's/>/On: /p
               s/</Off:/p'
```

Exercises

1. The command `sleep` "sleeps" for the number of seconds specified by its first argument. For example, the command `sleep 15` will cause there to be a 15 second delay between pressing return and the appearance of the next command prompt. Explain how `sleep` can be used in conjunction with the `echo` command to create an alarm clock.
   Hint: The command `echo ^G` will cause the terminal to "beep".
2. What happens when no files match a pattern on a command line? Hint: Investigate this using `echo`.