Basic Unix Scriptwriting

by Don Gingrich

For students in

UNIX Essentials for System Administrators
UNIX System Administration

Scripting Notes

UNIX Systems Administration
and
UNIX Essentials for System Administrators
RMIT University
School of Computer Science and Information Technology

July 27, 2014

Introduction

An introduction to shell script writing is a part of the new "UNIX Essentials for System Administrators" subject. This subject is one of two alternate prerequisite paths into "UNIX System Administration". Operating Systems does not provide the same introduction to scripting, but there is a necessary goal of avoiding unnecessary duplication of lecture material between courses. This text is intended to provide the material that will be in the equivalent lectures in UNIX Essentials. It should provide students from an Operating Systems background with the necessary scripting knowledge to be successful in UNIX System Administration.

In general, the style of this text is intended to be relatively conversational. It includes several examples of real world situations where particular scripts, or aspects of scripting were significant. The intent of these sections is to demonstrate the relevance of the related material.

Most system administrators will write scripts as a part of their job. There are a number of reasons for writing scripts. These reasons include the following:

- Scripts are *portable*. As long as the machine has an *interpreter* for the scripting language used, the script can generally be run with little or no modification.
- Write once, run many times. Once a particular problem is solved through the use of shell commands it is relatively simple to turn the list of commands used to solve the problem into a script. Then, if the problem occurs again, the script can be run.
- Scripts reduce typing. This is an advantage because less typing means less chance of an error. It's also an advantage because it means less work.

- Scripts can be used with `cron` to do system maintenance at times when the load is light. The `cron` daemon can be configured to run commands either repeatedly or at specific times. This means that system maintenance tasks can happen in the middle of the night to avoid inconvenience for users. There are specific additional considerations with the use of `cron` but they will be discussed later.

- Scripts can provide specialised commands for users. If users need to do a specific and moderately complex to very complex task for which a series of commands is required, a system administrator can write the required commands into a script an make the task easier for the users.

  An example of this was the script to delete semaphores and shared memory blocks during a recent Operating Systems assignment. Users needed to get the ID number of their blocks and semaphores and individually remove them. The following script did the whole task with a single command:

```
#!/bin/sh
# A short script to remove Inter-Process Communication primitives on
a Sun system
IPCS=/usr/bin/ipcs
GREP=/usr/bin/grep
WHOAMI=/usr/ucb/whoami
AWK=/usr/bin/awk
XARGS="/usr/bin/xargs -l"
IPCRM=/usr/bin/ipcrm

$IPCS -s | $GREP `$WHOAMI` | $AWK 'print $2' | $XARGS $IPCRM -s
$IPCS -m | $GREP `$WHOAMI` | $AWK 'print $2' | $XARGS $IPCRM -m
```

  Come back and look at this script later to see if you can work out what it is actually doing.

- Scripts are easier to debug. Studies have shown a relatively consistent value for the "number of bugs per thousand lines of code". This value is independent of the language used, *i.e.* it does not change much depending on whether the program is in C, Assembly, Pascal or shell-script. Shell scripts do significantly more work in an equivalent number of statements. See the side-bar comparison of versions of "Hello World" in shell and C as an example. Thus, for a given task, the shell script is likely to have fewer bugs.

Hello World

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{

    printf("Hello World \n");

}
```

versus

```
#!/bin/sh
echo "Hello World"
```

And the difference in numbers of lines only increases with increasing complexity.

In this text the primary topic is the use of Bourne Shell scripts. Bourne shell has several characteristics that make it particularly interesting. The Bourne shell was the second shell commonly distributed with early versions of AT&T UNIX. The version of the UNIX source code that Ken Thompson took to the University of California at Berkeley included a copy of the code for the Bourne shell. As a result, Bourne shell is present on essentially *all* UNIX systems, irrespective of whether they came from the AT&T or the BSD development stream. Linux systems also provide a shell that is similar to the Bourne shell. The BASH (**B**ourne **A**gain **SH**ell) is a GNU developed shell that supports the full Bourne shell command set. But, like most software from the GNU project, the BASH adds significant extensions to the Bourne shell. These extensions are not a problem so long as a script will only run on systems that have BASH installed. But attempting to run a script written for BASH on Bourne shell may cause significant problems due to the absence of required functionality.

A key point that needs to be made about Bourne shell is that Bourne shell will be found on *all* UNIX systems.

An example that hopefully demonstrates the universal nature of the Bourne shell is worth presenting.

In 1997 the author was working for a hardware and software company that specialised in sales to the retail sector. I was asked to write a script that the (Non-UNIX-literate) store manager could use to transfer the price database from the old machine to the new. The old machine was nearly 10 years old, used a different byte order for storing numbers, ran a different version of UNIX, and had non-standard RS-232 ports that required a custom cable to actually connect the machines. But the Bourne shell on these machines ran the same

script in the same way. This is what is meant by saying that Bourne shell is a universal constant in UNIX systems.

Note that, because the BASH command set is a "super-set" of the Bourne shell command set, a Bourne shell script "*should*", under most conditions work the same way under BASH as under a "real" Bourne shell. The "*should*" is because, unless every possible Bourne shell script was tested under BASH, it is impossible to be absolutely certain that all Bourne shell scripts work exactly the same under BASH. What this means for assignments is that, while it is perfectly acceptable to write scripts and do initial testing on a Linux system, the scripts need to be given final testing under Solaris or other UNIX to insure that they conform to Bourne shell syntax.

What about the `csh` or `tcsh` or `ksh` or ...?

The Bourne shell has, as one of its other advantages, a rich set of options for controlling the destination of output from commands and input to commands. Particularly when writing scripts for users, it is best to catch any error messages produced by commands called by the script. The script can then produce its own error messages, as required. These messages can be more specific and relate the error to the specific context of the script. This contributes to significantly better usability.

The other shells typically lack this flexibility of I/O re-direction. In particular, the `csh` and `tcsh` have problems in this area. It is for this reason that most writers recommend against the use of these shells for script writing.

Korn shell (`ksh`) is an exception with respect to problems with I/O and re-direction. The features in Korn shell are equal to the features of Bourne shell. Effectively, the Korn shell provides a "super-set" of the features of Bourne shell. The problem with the use of Korn shell is that it is not as commonly available as the Bourne shell.

In any case, where portability is not an issue it may be reasonable to use `bash`, `ksh` or any of the less common shells. But portability is a problem that often only becomes apparent substantially after the original script is written. If it needs to be moved to an incompatible system, (one that lacks the required interpreter) then it will probably need to be re-written.

The Basics

Hopefully the Introduction has demonstrated that learning to write shell scripts is useful and effective.

This section will introduce the basics of writing shell scripts. Simply put, a shell script is a series of statements that could be entered at a shell command line to do some task. A couple of key points here. The statements could be entered at a command prompt. A shell script uses a combination of commands that are "built-in" to the shell and external commands. An external command is a separate program that is "launched" by the shell using a combination of the "`fork()`" and "`exec()` functions. The shell "waits" for the result to be returned from the external program and can use the result to control the future flow of the script execution.

Interpreter

All scripts, whether they are Bourne shell, PERL, Python, or whatever are *interpreted* languages.

What does *interpreted* mean in this context and why it this an important distinction?

A program called an interpreter reads a script, line by line and follows the instructions on the line. This means that the actual content of the line, including commands, is not necessarily "fixed" until the time when the interpreter reads the line. As we will see shortly, this makes interpreted language programming substantially different from programming in compiled languages.

The interpreter for a script is specified using a special construct on the first line of the script.

Scripting languages all recognise a line that begins with the # (hash or sharp character) as a comment line. The operating system always looks at the first few characters of an executable file to determine how to load and run the file. For example, a standard, dynamically linked binary file always begins with a byte with value 0x7f, followed by the characters ELF, an abbreviation of (**E**xecutable and **L**inking **F**ormat). And scripts always begin with `#!` followed by the full path of the interpreter. On most systems the Bourne shell is `/bin/sh` so a Bourne shell script will have `#!/bin/sh` on the first line. In contrast, PERL is commonly `/usr/bin/perl` so a PERL script will begin with `#!/usr/bin/perl`.

For what it may be worth, the PERL developers are fond of puns. As a result, they looked at the the beginning two characters of the line (`#!` and saw the "`#`" as a "sharp" and the "!" has been commonly referred to by `Unix` users as a "bang" for some time. So the combination became shebang. Incidentally, there is a US expression that refers to "the whole shebang" in a completely unrelated context. So, if you hear someone refer to the the start of a script as shebang, you'll know what they are talking about.

Comments

Comments in a script begin with the `#` character. The `#` can be anywhere in a line. Any characters after the `#` are ignored by the interpreter. If a script absolutely *must* process data that contains the `#` character, there are ways of "escaping" it so that it is ignored.

You will note that the interpreter line above is, in fact, a comment. This means that it is safe to type something like:

`$ /bin/sh` *scriptname*

The shell will simply ignore the "interpreter" line since it is a comment, and the script will run as expected.

Deciding how to comment a script and how many comments need to be included is difficult for many people. At a minimum, a comment at the top of the script that describes the purpose of the script, lists possible portability considerations, names the author of the script, provides a basic revision history for the script and lists any relevant copyright information is required.

If a script uses internal functions, these functions should have similar information to the script information.

Beyond this level, there are differences of opinion about the required level of commenting. One argument holds that scripts are designed to be "self-documenting" and few, if any, further comments are required.

An alternative view is "comment everything" "just in case".

A sensible level of comments should include discussions of any complex constructs that might be confusing to someone seeing the script for the first time. This also means that if you write a script and come back to it a year later, the comments will be there to explain why particular options were chosen. This is particularly relevant when there was an "obvious" way to do something that simply did not work.

Avoid writing "job security" scripts. (Job security scripts are scripts that use obscure and obfuscated code to make it difficult or impossible for anyone else to modify or maintain them.) This approach does not particularly impress employers. Also, there may come a time when you want to be promoted internally within the company and don't get the promotion because, "Who could maintain those essential scripts if Fred isn't here?" Job security code is seldom a good idea in the long run.

Variables

Variables in shell scripts are an interesting aspect of how scripts work. In Bourne shell a shell variable is defined using the form:

```
FOO=bar
```

This sets the value of the shell variable `FOO` to `bar`. Note that there are no spaces on either side of the "`=`" sign. Putting spaces in this line will cause problems. To use the variable, use the form:

```
$FOO
```

The line:

```
echo $FOO
```

will produce output of:

```
bar
```

Shell variables will be concatenated in lines, but if you tried to use the line:

```
echo $FOObell
```

thinking that the output would be barbell, you would be disappointed since no output results. Why does this happen? The problem is that the shell looks for a variable named FOObell which does not exist. See the following sequence:

```
$ FOO=bar
$ echo $FOObell

$ FOObell=gotcha
$ echo $FOObell
gotcha
$ echo ${FOO}bell
barbell
```

As you can see, the solution is to enclose the variable name in "curly braces" ({}). Curly braces are *required* when a line is concatenating the value of a variable with some other text. Curly braces are *optional* at any time.

In the **Advanced Topics** section we will look at some further constructs with the variable name and curly braces that allow the script writer to set default values for variables and other useful options.

The contents of a variable can be a number, a character string (which can include spaces), or the name of a program, possibly with options specified.

To specify a shell variable that includes spaces, enclose the value(string) in quotes. Otherwise, the shell will interpret the whitespace (a term for the characters space, tab and return) as a delimiter (a separator).

An example of using a shell variable to represent a command follows:

```
$ LS="/bin/ls -l *.tex"
$ $LS
-rw-r--r--   1 gingrich staff       7835 Dec  7 17:14 a_file.tex
-rw-r--r--   1 gingrich staff      48466 Oct 31 13:23 a_story.tex
-rw-r--r--   1 gingrich staff      54297 Dec  6 13:16 another_file.tex
-rw-r--r--   1 gingrich staff      46529 Dec  6 13:05 some_file.tex
```

In this example `$LS` represented the command `ls` with a specific set of command options. This can be useful in scripts that need to be portable. By specifying a particular version of a command, a script author can be sure of the correct options being available and behaving in the expected manner.

As an example for the reason for this, consider that there are three versions of the relatively simple `ls` command on the Computer Science machines, `yallara` and `numbat` which use the Sun Solaris operating system. There is a version in the directory `/usr/bin` (the standard Solaris AT&T System V version), a version in `/usr/ucb` (the Berkeley or BSD version), and in `/usr/xpg4` (the POSIX standard version).

The obvious question is, "Why does Sun provide so many different versions of this command?"

The answer is relatively simple. The first operating system distributed by Sun was named SunOS and was based on the BSD operating system. In the early 1990s there were general complaints from both Unix users and software vendors writing applications for Unix. The principle complaint was that there were so many varieties of Unix that finding a single "platform" for applications was difficult.

Two things came out of this push One was a consistent set of API functions (**A**pplication **P**rogramming **I**nterface) functions for X-Windows and general adoption of X-Windows as the graphical interface for UNIX systems. The other was the development of the POSIX standard API for UNIX.

To make it easier to implement the POSIX API, Sun purchased the AT&T UNIX source code and modified it slightly to produce the new "Solaris" operating system.

But now Sun had a "new" operating system that was not completely compatible with the earlier, BSD based system. What could they do to protect the investment of customers who might have written extensive scripts that relied on specific BSD behaviour in commands? The solution has been to provide the `/usr/ucb` directory that contains the BSD commands where they are different from the Solaris (SysV) versions. And for scripts written to the POSIX standard, when the POSIX command differs from the SOLARIS command, the POSIX version is in `/usr/xpg4`.

Also, last, but not least, if a system administrator wants to use the GNU versions of particular commands, the source can be downloaded and compiled and the compiled version stored in `/usr/local/bin`. Note that GNU utilities typically have extensions to the command options compared to all of the other versions of a command that may be available.

What does all of this have to do with shell variables?

A good question...

The problem with all of these different command versions in a Solaris system is that there is always a question about which command version is going to be used. If a script depends on a particular command syntax or output characteristics then there must be some way to ensure that the *right* version of the command is the one that is used. The obvious and probably wrong answer is to use the `PATH` variable. If all of the commands were in the same directory, the a `PATH` would work. But what if one command is in each of the possible locations and there are alternate versions in directories earlier in the `PATH`?

The solution is to use shell variables in the script to specify the commands that are to be executed. The convention is to name the variable the same as the command name, but in all UPPERCASE. The variable contains the "fully-qualified" or absolute path to the particular command. In the example above, that is `/bin/ls` for the `ls` command.

This will be mentioned again in the discussion of scripting standards.

Shell Variables and Environment Variables

Aren't shell variables and environment variables the same thing? They are actually subtly different. A shell variable only exists within the "current" shell. If a shell launches a command to do some task, the command will not be able to access the values of any shell variables. But, if a variable is an environment variable, the command will be able to access the value in the variable. This is especially important in the case of sub-shells. In many cases, the sub-shell needs to have the same variables available as the primary shell that the script is running in.

So what is a sub-shell? If a script calls a script function, the script function runs in a sub-shell. Sometimes loop code runs in a sub-shell. If a loop behaves in an unexpected manner, check to make sure that the variables are visible inside the loop.

How does a variable become an environment variable? The short answer is that it depends on the shell. For Bourne shell, a shell variable is created as above. To convert a shell variable into an environment variable, use the `export` command. As in:

```
$ FOO=bar
$ export FOO
```

Now `FOO` is an environment variable.

Is there a way to pass environment variables back to the parent? Not directly, but it is possible to run a script in the current shell which means that the variables that are changed are changed in the parent. Consider the following:

```
$ cat footest
#!/bin/sh

FOO=baz;export FOO
$ ls -l footest
-rwx------   1 gingrich staff         30 Dec 12 18:23 footest
$ FOO=bar
$ echo $FOO
bar
$ . ./footest
$ echo $FOO
baz
$ FOO=bar
$ echo $FOO
bar
$ ./footest
$ echo $FOO
bar
```

Let's look at what happened. The script `footest` is simple. It sets the value of the variable `FOO` to `baz` and makes `FOO` an environment variable. The next line sets `FOO` to `bar`. Then we check to see that `FOO` really is `bar`. It is. The next line is where the "tricky" stuff happens. The construct is `. ./scriptname` and it means, "run this script within the current shell". When we echo `$FOO` now, the result is the value that was set by the script. If we go back and repeat the experiment, but run the script normally, without the leading dot "." the script does not change the value of `FOO` in a way that is permanent.

Quotes and What They Mean

To begin with consider that there are three types of quotes on your keyboard.

- The quote type referred to as a "Single-quote" is located next to the enter key and is typed without the shift key – it looks like this `'`.

- The quote type referred to as a "double-quote" is located next to the enter key and is typed with the shift key – it looks like this ".
- The quote type referred to as a "back-quote" or "back-tic" is generally located in the upper left of the keyboard next to the "1" key – it looks like this ‘.

OK then, here are some examples.

```
    1 $ foo=text
2 $ echo $foo
text
3 $ echo '$foo'
$foo
4 $ echo "$foo"
text
5 $ bar=stuff
6 $ echo $bar
stuff
7 $ echo $foo
text
8 $ bar=‘echo $foo‘
9 $ echo $bar
text
10 $ baz="Some stuff with spaces"
11 $ echo $baz
Some stuff with spaces
12 $ baz=Some stuff with spaces
13 $ echo $baz
Some
```

Right then, what happened?

In line 1 we set the shell variable foo to be equal to text. Then in line 2 we echo the text to make sure that it is there. in the following unnumbered line the output of text is printed.

In line 3 we look at what happens with single quotes. Note that what is printed is what was on the command line. The shell variable is not expanded. This can sometimes be useful if you need to pass a shell variable to a command within a command.

In line 4 we put `$foo` in double quotes. Note that within double quotes the shell variable is expanded.

In line 5 and 6 we are creating and testing another shell variable. So now `bar` or `$bar` is set to stuff.

In line 7 we are checking to make sure that `foo` is still `text`. Then in line 8 we use the back-tics to assign the result of the echo command to the variable `bar`. Remember that

`bar` was `stuff` before line 8. And when we test in line 9 we see that `bar` now holds text, just the same as `foo`.

In 10 through 13 there is another example of what double quotes are used for. In 10-11 we set `baz` to a group of words with spaces. The double quotes mean that the shell interprets what is between them as a single argument. (Normally the space is what is referred to as a delimiter meaning that a space separates arguments. The double quotes stop this from happening.)

Note that in lines 12 and 13 without the space only the first word is assigned to the shell variable.

How the Shell Handles the Lines of Input

On the initial consideration, the title of this section may seem a bit silly. After all, the shell reads a line form the script or the command line and then executes it, right? The answer to that is, "Yes, the shell does read a line and execute it, but what happens in the cases where there are shell variables, quotes and parenthesis in the input line?"

The reality is that there is a hierarchy in the order of interpretation of the input line. The shell starts at the inmost level of quotes or parenthesis and substitutes the values for shell or environment variables. If there is a set of back-tics, the command or commands within the back-tics are executed and the result is placed within the command line in the space marked out by the back-tics. Then the next higher level of the line has any remaining substitutions made. The commands at this level are run. Eventually the shell reaches the top level and runs the entire line.

A simple example:

```
#!/bin/sh
SED=/bin/sed
CP=/bin/cp
foo=filename.c # note that this would usually be set
# in a for statement or equivalent
$CP $foo `echo $foo | $SED 's/\.[a-Z0-9]*/.bak/'`
```

What will happen?
The contents of the back-tics will be interpreted first. So the contents of the back-tics become:

```
echo filename.c | /bin/sed 's/\.[a-Z0-9]*/.bak/'
```

The result of executing these commands is `filename.bak` So now the command line is:

```
/bin/cp filename.c filename.bak
```

When this command runs a backup copy of the file is made. Note that the command in `sed` *must* be in the single quotes, otherwise the shell would try to interpret the regular expression that is being passed to `sed`.

Note: The `echo` command is built into the shell and thus does not need to have an absolute pathname.

Flow Control

`if` Statements

The basic construct of an `if` statement in Bourne shell is:

```
if [ $var = "condition" ]
then
    do something
fi
```

If the comparison of the variable var with condition is true then whatever "something" is will be run by the script.

This sort of `if` statement will manage the requirements of a substantial percentage of scripts, but the statement can be significantly more complex, *e.g.*

```
if [ $var = "condition" ]
then
    do something
elif [ $var = "condition2" ]
then
    do something else
else
    do something completely different
fi
```

In the full construct, it is possible to have many `elif` statements. But there can be only one initial `if` statement and only one `else` clause at the end. In the above construct, the flow of execution would would proceed through testing the various options until either an option was true *or* the `else` clause is reached. Whenever an option is true or the `else` clause is reached, the associated commands are run. None of the other commands are run and execution continues with the line following the `fi` statement.

There is a final alternative way of writing an `if` statement that is often used. In this variation the semi-colon "`;`" is used to replace the return between the end of the test and the then statement. It looks like this:

```
if [ $var = "condition" ] ; then
    do something
elif [ $var = "condition2" ] ; then
    do something else
else
    do something completely different
fi
```

The advantage is that it takes up a bit less space and the relationship between the `if`, `elif` lines , the `else` and the `fi` are much clearer.

In your script writing, choose one of the alternatives and stick to it consistently. Do not switch back and forth between the styles since this will reduce the readability of your script.

Square Brackets

The square brackets are a shorthand way of using the `test` command. Look at "`man test`" for details of the options for the command and the tests that it will perform. The square bracket shorthand was developed to allow people who are used to normal programming languages to use `if` statements that look what they are used to from other languages. **But** it is important to remember that when using the square brackets (`[ ]`) there must be a space after the leading `[` and before the trailing `]` or the shell will not interpret them correctly. The reality is that there is actually an executable file with the name "`[`" on Linux systems.

```
/usr/bin # ls -l test [
-rwxr-xr-x 1 root root 27248 2006-04-23 16:48 [
-rwxr-xr-x 1 root root 24388 2006-04-23 16:49 test
```

This is not true on a Solaris system. On Solaris, the "`[`" construct is provided by the shell.

The simplest option to remember is to *always* put a space on either side of *both* of the square brackets. It also makes the script easier to read which is an advantage in any case.

The `test` command will return zero (0) for true and non-zero for false. This is exactly opposite to the way that C language programs and may be a source of some confusion to some people. Why does UNIX do this if it confuses people? The simple answer is that true is equivalent to successful completion of a program. And, it should be clear that, in most cases success is what happens when the program terminates normally. Usually there is only one way that can happen. But there are potentially lots of ways that a program can terminate *abnormally*. So, since zero is unique, it is reserved for normal termination, OK or true. And non-zero becomes abnormal termination, Not-OK or false.

In the section titled, Commands Useful in Scripts, there will be a more detailed discussion of the test command and the available options.

`case` Statements

The `case` statement is effectively just a different way of writing the `if`, `elif`, ..., `else`, `fi` series of statements. The syntax is similar to the syntax of a C language `switch` statement. A case statement consists of a series of patterns to be matched, with each pattern followed by a right parenthesis ")". This is followed by a series of statements, one per line. Finally, the series of statements is concluded with two semicolons. Unlike the C switch statement, the case statement does not allow "fall-through" by leaving out the "end of option" double semicolon marker.

The pattern to be matched can include the '*' which matches any series of characters, '?' which matches any single character, and the pipe '|' character which acts as an "or" to select on either of two patterns. Taking the previous example:

```
if [ $var = "condition" ] ; then
    do something
elif [ $var = "condition2" ] ; then
    do something else
else
    do something completely different
fi
```

In `case` statement form this would look like:

```
case $var in
condition)
    do something
    ;;
condition2)
    do something else
    ;;
*)
    do something completely different
esac
```

A further example with some of the more complex conditions:

```
#!/bin/sh

echo "test for $1"

case $1 in

   f?t|g*)
          echo 'f?t|g* option selected'
          echo "$1 was entered"
          ;;
   foot)
          echo "foot option"
          ;;
   b*)
          echo 'b* option selected'
          echo "$1 was entered"
          ;;
   *)
          echo default
esac
```

The output looks like this:

```
> ./case-test fot
test for fot
f?t|g* option selected
fot was entered
> ./case-test foot
test for foot
foot option
> ./case-test fool
test for fool
default
> ./case-test gaz
test for gaz
f?t|g* option selected
gaz was entered
> ./case-test gaol
test for gaol
f?t|g* option selected
gaol was entered
> ./case-test zot
test for zot
default
```

`for` Loops

The `for` loop is a common construct in shell scripting since a lot of scripts are written to process a list of items. The script `for` loop behaves very differently from the for loops in most programming languages. Programming language for loops typically use an "index" variable that is incremented. Script for loops have a list of items to be processed. as in:

```
for item in aa bb cc dd
do
    echo $item
done
```

Will produce output of:

```
aa
bb
cc
dd
```

The generic form of the statement is:

`for` *item* `in` *list*
`do`
    do something
`done`

The "*list*" in the "`for`" line can be any list of one or more items. Since a shell or environment variable can contain multiple strings separated by "whitespace" a shell variable can be used here. The *something* in the loop is a statement or a list of statements.

Whitespace - What is it?

Whitespace is the technical term used to describe either space, tab, or return characters. Note that in UNIX a return character is actually the line feed character, `0x0a`.

A common problem that UNIX novices experience is editing files in a Windows editor and having difficulties running, compiling, *etc.* in UNIX. The reason for this problem is that Windows, and DOS before it, use a combination of the carriage return character, `0x0d` and the line feed character. These "extra" carriage return characters can be seen as "`M̂`" characters at the ends of the lines in a `vi` editing session.

`while` loops and `until` loops

If a script requires the equivalent of a C language for loop, how can we create one? Creating a loop with a counter is fairly easy using a `while` loop.

```
counter=1
while [ counter -lt 5 ]
do
    echo counter is $counter
    counter=‘expr $counter + 1‘
done
```

The variable `counter` is a standard shell variable. And the increment is handled by the `expr` command. (Note that `expr` is an external command, *i.e.* not part of the shell. The `expr` command in Solaris is available in two forms, the System V form in `/usr/bin/expr` and the BSD form in the file `/usr/ucb/expr`)

The contents of a while loop will be run none or more times depending on the comparison. An `until` loop is defined in much the same way that a `while` loop is defined. The primary difference is that an `until` loop tests at the *end* of the loop. Thus, the contents of an `until` loop will *always* run at least once.

To re-write the above `while` loop as an `until` loop the result would be:

```
counter=1
until [ counter -ge 5 ]
do
    echo counter is $counter
    counter=‘expr $counter + 1‘
done
```

Depending on the purpose of the script, one or the other of the above loops will be the better choice. Note that they do not need to use numeric counters. An `until` loop might be an ideal choice of the loop to control a menu driven interface, as in:

```
choice=""
until [ "choice" != "q" ]
do
    code to display menu
    read choice
    case $choice in
        1) do something
           ;;
        2) do something else
           ;;
        q|Q)  choice=q
           ;;
        *) echo invalid choice entered
    esac
done
```

Statements

I/O Re-Direction

Input-output redirection is one of the key features that distinguished the Bourne shell and its derivatives such as the Korn shell, BASH and others from the C shell (csh) and its derivatives such as the `tcsh`. I/O redirection is useful in UNIX because of the basic design of utility programs. UNIX programs are designed to, by default, accept input from `stdin`, send normal output to `stdout`, and send error output to `stderr`. This design allows for programs to be either linked together or to have their output placed into a file.

I/O Redirection Symbols	
Symbol	What it does
`>`	Without any additional optional characters redirects the output to a file. The existing contents of the file are deleted.
`<`	Redirects the contents of the file on the right of the symbol into the program on the left side.
`|` (pipe)	Redirects `stdout` of the program on the left to `stdin` of the program on the right. (Both sides *must* be executable programs.
`>>`	000
`<<`*delimiter*	
`2>`	Redirect `stderr` to a file. The redirection works the same as the simple `>`
`>`*file* `2>&1`	

Notes:

1. The file is emptied by the shell before the command line is actually executed. Thus, a command line of `cat foo > foo` would result in an empty file named foo.

The pipe, "`|`" connects `stdout` from the program on the left side to `stdin` of the program on the right side.

The Next Level

This section will consider several more advanced topics in scripting. Debugging, efficiency, and advanced variable manipulation are included.

Debugging Your Scripts

As you write more complex scripts the time will come when they do not work correctly. Sometimes the problem will be obvious from the error message provided by a program called by the script. Often the problem will be nearly impossible to determine. One solution in debugging is to use `echo` or `printf` statements to display the contents of variables and possibly the expected value or the line number of the statement. This is a debugging method that has been used for many years. The trick is making sure that there is a print statement both before and after any possible branch in the execution of the script.

Another alternative in script debugging takes advantage of the way that the shell actually works. As the shell interprets the line of input, you can ask the shell to display the staged interpretation.

Returning to our example of a multi-level script presented earlier:

A simple example:

```
#!/bin/sh -x
SED=/bin/sed
CP=/bin/cp
foo=filename.c # note that this would usually be set
# in a for statement or equivalent

$CP $foo `echo $foo | $SED 's/\.[a-Z0-9]*/.bak/'`
echo This is an output line
```

We added `-x` after the `#!/bin/sh` to tell the shell to show us the steps in the execution. The output looks like this:

```
+ SED=/bin/sed
+ CP=/bin/cp
+ foo=filename.c
++ echo filename.c
++ /bin/sed 's/\.[a-Z0-9]*/.bak/'
+ /bin/cp filename.c filename.bak
+ echo This is an output line
This is an output line
```

Note that some lines have two plus signs '+' in front of them and others have only one sign. The number of plus signs tells us how many levels deep we are in the interpretation. Lines with more pluses happen before lines with less pluses. The comments are not produced in the output.

But this output has a lot of lines that we probably do not care about. The statements assigning values to the shell variables are a bit of a wast of time. There is a way to only get the full interpretation of a part of a script.

If we mark the start of the full interpretation with a line that says:
`set -x`
and the end with a line that says
`set +x`
we can limit the amount of output we get.

As an example consider:

The same simple example:

```
#!/bin/sh
SED=/bin/sed
CP=/bin/cp
foo=filename.c # note that this would usually be set
# in a for statement or equivalent

set -x
$CP $foo ‘echo $foo | $SED ’s/\.[a-Z0-9]*/.bak/’‘
set +x
echo This is an output line
```

We added `set -x` before the line we wanted to see and `set +x` after the line we wanted to see. Of course, there could be any number of lines between the `-x` and the `+x`.
Now the output looks like this:

```
++ echo filename.c
++ /bin/sed ’s/[a-Z0-9]*/.bak/’
+ /bin/cp filename.c filename.bak
+ set +x
This is an output line
```

Note that this version omits the assignment statements and does not display the `echo` statement, only the output from the echo statement.

Efficiency

Realistically, if efficiency is the primary concern, it is unlikely that any scripting language will be the language of choice. Scripts simply are not efficient since they need to load and execute multiple programs as they run. But, even considering the inefficiency of scripts, it is possible to look at ways of at least minimising the inefficiency.

When designing the program flow in a script, try to avoid multiple nested loops that result in $O(n^2)$ or worse performance. There is generally a way to do this.

When designing a script, it is generally better to use a series of steps linked with pipes rather than sending output to a file and then reading it back into the script. Pipes are fast, since the transfer is in memory. File I/O is slow since disk access is involved.

As an alternative to file I/O consider using shell variables. Shell variables are also held in memory and will be a faster alternative than file I/O.

Be careful to avoid multiple processing of the same data, if possible. As an extremely bad example of how not to do things, there was a script run by a researcher on goanna

a few years back that was processing one line at a time from a file that contained several million lines.

The critical part of the script looked something like the following code:

```
LENGTH=‘cat filename | wc -l‘
LineNO=1

while [ $LineNO -le $LENGTH ]
do
    LINE=‘head -$LineNO filename | tail -1‘
    echo "Line number $LineNO is ==>${LINE}<=="
    LineNO=‘expr $LineNO + 1‘
    # do stuff with the line
done
```

If you are interested, this script will print a file, one line at a time. That was the reason for including the echo line. The problem with the script on a multi-million line file is that it re-reads the entire file for each line of input to the script. Re-reading and discarding millions of lines of a text file takes a l-o-n-g time.

Unfortunately there is no easy fix for this in standard shell scripting.

The obvious solution that is wrong looks like this:

```
LENGTH=‘cat filename | wc -l‘
LineNO=1

while [ $LineNO -le $LENGTH  ]
do
        read LINE < filename
        echo "Line number $LineNO is ==>${LINE}<=="
        LineNO=‘expr $LineNO + 1‘
        # do stuff with the line
done
```

Using the following file for input:

```
This is line 1
This is line 2
This is line 3
This is line 4
This is line 5
This is line 6
This is line 7
This is line 8
This is line 9
```

The output looks like this:

```
Line number 1 is ==>This is line 1<==
Line number 2 is ==>This is line 1<==
Line number 3 is ==>This is line 1<==
Line number 4 is ==>This is line 1<==
Line number 5 is ==>This is line 1<==
Line number 6 is ==>This is line 1<==
Line number 7 is ==>This is line 1<==
Line number 8 is ==>This is line 1<==
Line number 9 is ==>This is line 1<==
```

This does not solve the problem. The reason that it does not work is that the file is opened on each iteration of the loop. A non-obvious point to consider is that, in shell scripting, a while loop or for loop effectively runs in a sub-shell. The solution to the problem is to look at how we can take advantage of the sub-shell.

A script that *will* correctly process the file looks like this:

```
LENGTH=‘cat filename | wc -l‘
LineNO=1

while [ $LineNO -le $LENGTH  ]
do
        read LINE
        echo "Line number $LineNO is ==>${LINE}<=="
        LineNO=‘expr $LineNO + 1‘
        # do stuff with the line
done < filename
```

Since the file is input to the loop, the read inside the loop reads a new line on each iteration. The output looks like this:

```
Line number 1 is ==>This is line 1<==
Line number 2 is ==>This is line 2<==
Line number 3 is ==>This is line 3<==
Line number 4 is ==>This is line 4<==
Line number 5 is ==>This is line 5<==
Line number 6 is ==>This is line 6<==
Line number 7 is ==>This is line 7<==
Line number 8 is ==>This is line 8<==
Line number 9 is ==>This is line 9<==
```

Advanced Variable Referencing

In referencing variables, it may be useful to utilise the following forms which allow setting default values. In several past assignments there has been a requirement to use environment variables to allow the writing of a script that would attempt activities that, while they are appropriate for an administrator, are not allowed for ordinary users on the School of Computer Science and Information Technology machines. An example of this would be a script designed to modify a configuration file, the passwd file or the shadow file.

To write and test these scripts, students were required to use an environment variable, such as PASSWD to point to the file. This also made testing of the scripts easier for the tutors. But, there was also supposed to be a default value of `/etc/passwd` for example in the case of the passwd file.

So, how can this be done easily? Quoting from the Solaris manual page, the forms are:

${*variable*} This form of a variable name is always acceptable but it is only *required* when the desired output is the contents of the variable are to be catenated with some other text. The shell would have no way of determining where the variable name ends and the further text begins without the "curly brackets". ('{}')

${*variable*:-*default*} This is the most basic form for setting a default. If the variable is set and has a non-null value then the value will be used. Otherwise the specified default value is used. The value of the variable "*variable*" remains null if it was un-set. In the above example we could use something like:

```
my_passwd=${PASSWD:-/etc/passwd}
```

And then use the shell variable my_passwd everywhere in the script.

${*variable*:=*default*} In this form, the value of the default is substituted. This is different from the previous example since, after this statement, if the previous value of "*variable*" was null the new value will be the supplied default. Thus later references to "*variable*" in the script will find the value that was set, or the original value if there was one.

${*variable*:?*default*} here, if the variable "*variable*" does not have a value, an error will result.

${*variable*:+*value*} if "*variable*" already has a value, then *value* is substituted. Otherwise nothing is substituted.

A simple script to test these substitutions looks like this:

```
#!/bin/sh

var1=${1:+foo}
echo "var1 is ${var1} \$1 is ${1}"

var2=${zot:=bar}
echo "var2 is ${var2} \$zot is ${zot}"

var3=${2:-default_value}
echo "var3 is ${var3} \$2 is ==>$2<=="

var5=${2:?"No second command line option"}
echo "This line is not reached with no 2nd option"
```

The output looks like this:

```
> ./variable-test tester
var1 is foo $1 is tester
var2 is bar $zot is bar
var3 is default_value $2 is ==><==
./variable-test: line 12: 2: No second command line option
```

Shell Built-Ins

Most shells have a number of "built-in" commands. In the Bourne shell, there are a number of built-in commands for flow control which have already been discussed. The `cd` or `chdir` ((change directory) command, the `echo` command, `eval`, `exec`. `exit`, `getopts`, `read`, `return`, `shift`,`test`, and `trap` are all potentially useful in scripts. There are several other built-in commands not in the above list. Use the command "`man -s1 sh`" on yallara or numbat to see the full manual page for the Bourne shell. Note that some of these commands also exist as executable programs. In many cases the "stand-alone" executable programs may have more options than the built-in command offers. The built-in will run more quickly and with lower resource requirements, but if the executable provides required functionality, it is generally worth the cost.

Of course, there are commands that realistically *must* be part of the shell to function correctly. The `cd` command is an example of this type of command. The current working directory is part of the environment, *i.e.* an environment variable. If the `cd` command is an independent executable what will happen? When the command is typed at the command prompt, the shell uses a `fork` function, followed by an `exec` function to load and run the command. But, because the command runs as a child process, any changes that it makes to the environment disappear when the child terminates. See the section, "Shell Variables and Environment Variables" for a demonstration of this problem.

Other commands that do not modify the environment, can effectively provide greater functionality with an independent executable program.

The next questions is how to control which version of the command is actually run. The answer is fairly simple. If you either type the command, just the command, either at a command prompt or in a script the built-in version will be used. If you type a command with either an absolute or relative path, the actual executable will be used. (This, of course, assumes that the file exists and that it is executable.)

Standards in Scripting

Corporations that use scripts as a part of an integrated computer system will typically have a series of *standards* to ensure that the scripts are as maintainable and portable as possible. These standards may include rules for comments, rules for variable names, layout (indentation, where the `then` in an `if` statement should be placed, etc. This section will include information about some common standards.

Most commercial scripts begin with a big comment. The comment includes, among other information, the company name and copyright information, the author of the script, what the script is designed to do, the expected input parameters and output of the script, and a revision history. If the script is interactive, the input and output information may be omitted.

Corporate scripting standards will often also specify that the commands used in the script should be enumerated in a block of variable definitions of the form:

```
LS=/usr/bin/ls
```

Note that the variable name is the command name, but in uppercase. This convention is commonly part of the standard. Also note that the command is specified as an absolute or "fully-qualified" pathname.

If the goal was writing an interactive script, one consideration is limiting or eliminating extraneous output from commands used within the script. Error messages from commands within the script could be confusing to a user. An example of this is the `rm` command. If the script is designed to remove a temporary file that might not be created in some circumstances, then the user could be presented with a "file not found" message that does not clearly identify which file was not found. Also, some versions of `rm` may question removing a file that is marked "read-only". The solution is to define the `RM` variable as the command plus a switch, as in:

```
RM="/usr/bin/rm -f "
```

Now using `$RM $foo` in a script will remove the file if it exists and will not complain if it is read-only or if the file doesn't exist.

The other major advantage of this approach to commands is that, if the script is "ported" to a different UNIX or UNIX-like platform that has some commands in different directories, the script will need to be changed only once per command, since the actual command is defined in a single location.

Another look at comments

Comments are probably the most contentious single consideration in standards. How much is "enough" commenting? How much is "too much" commenting?

There's no absolute rule for comments, but a few guidelines are worth considering. If a line of code uses an unusual regular expression to select data or to make some sort of decision, it is probably worth commenting what is being selected in some more easily readable form. If the code uses an uncommon feature of a program or utility, a comment explaining the feature is worthwhile.

But, at the same time, do not comment the obvious:
e.g `foo=‘expr $foo + 1‘ # increment foo`
This comment simply gets in the way of reading the flow of the script and probably makes it somewhat less readable.

Writing "job-security" scripts is not a good idea in any organisation. (In this context, a "job-security" script is one that is written in such an obfuscated style that it will be difficult or impossible to maintain for anyone other than the original author.)

There may also be a specification for variable naming. This will probably include the previously mentioned standard of using variables for commands and naming them with the command name in all uppercase. It may also include specifications for the names of environment variables and local variables within the script. As an example, local variables may be all lowercase and environment variables all begin with uppercase or possible use embedded uppercase for multi-word names. (*e.g.* FirstItem)

Regular Expressions

Writing regular expressions (or **regex**s) is the key to several aspects of shell script writing. This is a **big** topic. (O'Reilly has an entire book that discusses nothing but regular expressions.) So, you may consider these comments to be purely introductory in nature.

We will begin the discussion of regular expressions with a discussion of the "globbing" characters used in the shell. These form a limited sub-set of the regular expressions used by other UNIX tools such as `grep`, `sed`, and `awk`. The `perl` language uses even more complex regular expressions, which were derived, in part from the expressions used by `awk`. (If you are a `perl` purist, don't take offence at the above statement. The reality is that there were several similar but more limited systems around when Larry Wall began work on `perl`. It only made sense to build on previous work.)

One of the more unfortunate aspects of regular expressions is that each of the tools that use regexs has slightly (some might say annoyingly) different set of semantics and options. As a result, it would be a *very* good idea to avoid doing anything irreversible to files, etc, selected with regexs without first testing the expression by listing the output. This is a bit more difficult within a script. One solution is to "wrap" possibly destructive instructions in an `echo` statement. As in: `echo "$RM regexp"`

Suppose we wanted to remove all of the files with the " ~ " character in the filename.

(Note: some editors mark temporary files used for crash recovery with a " ~ " character in the name.) Most UNIX systems do not have any easy way to recover deleted files so an administrator using the `rm` command needs to be careful to *only* delete unwanted files. To remove these temporary files safely the best approach would be to first test the regular expression with the command: `ls *~*`
The expression `*~*` matches zero or more characters or any type (alpha or numeric or punctuation), followed by a tilde (`~`) then zero or more characters. Then, look at the output and make sure that *all* of the listed files need to be removed. Make sure that none of the files that are to be saved are in the listing. Then use the command:
`rm *~*`
Clearly, the bigger the directory is and the more complex the regex is, the more useful this technique will be.

Looking at the basics of UNIX shell globbing expressions, the '*' can represent none or many characters. As for example, "`ls *`" will list all of the files in a directory with the exception of "hidden" files. (Files with names that begin with a full-stop, or period or dot. All names for the same character, the un-shifted character that shares a key with >. Most sysadmins call this a "dot".) **Note:** In UNIX, unlike Windows, there is no need to specify "*.*" to list files with a basename and extension. As far as UNIX is concerned, a dot is "just another character" in most parts of a filename. The exception to this rule is that file or directory names that begin with a dot will not appear in a normal directory listing. (They are considered "hidden" files or directories.)

The '?' character will match any single character. If, for example, the goal was listing all files with names three characters long, then the expression "???" will work.

A slightly more complex problem is writing a regex that *will* list the hidden files. (OK, the easy way is the command, "`ls -a`". But `ls` provides a safe way to test general shell regexs.) Getting back to the regex, the statement "`ls .[!.]*`" will do the job in Bourne shell. Try it in your home directory on yallara. Most home directories have a significant number of hidden files so you should have some results.

14cm **Note:** One of the problems with UNIX as an "open system" is that a variety of people have developed UNIX commands, shells and other tools. There is no corporate "big brother" making sure that all of the commands work the same way. The above construct is a case in point. In Bourne shell and Korn shell the correct usage is as above:

```
ls .[!.]*
```

But, for `tcsh` (the default login shell on CS&IT systems), `csh`, `zsh`, BASH and probably a few other shells that use command history systems, the "!" character is used to signify a substitution from the command history.

On these shells, to get the same result you would need to use:

```
ls .[^.]*
```

How did that expression work? We wanted to list files that begin with a dot (".") so the leading dot in the expression matches the leading dot in the filenames. But, to avoid also listing all of the files and directories in the current directory (".") *and* all of the files and directories in the parent directory (".."), we need to specify that we only want files or directories that begin with a dot and then have something that is "not a dot" as the second character. The `[!.]` specifies anything that is not a dot. The "square brackets" enclose a group of characters. There are rules for specifying what is within the square brackets. A useful rule is that if the expression inside the square brackets begins with the exclamation symbol, "`!`", then the expression will match anything *except* the character(s) that follow the exclamation. The trailing star "*" specifies none or more characters following the "not dot" character.

If we had a directory that contained a book, with one file per chapter, the filenames might look like:

```
$ ls
chapter01.tex    chapter04.tex   chapter07.tex   chapter10.tex
chapter02.tex    chapter05.tex   chapter08.tex   chapter11.tex
chapter03.tex    chapter06.tex   chapter09.tex   chapter12.tex
```

Suppose there was a need to do some processing on chapters 1-5, 11 and 12. A `for` loop would handle the processing, but how can the required files be specified?

```
$ for file in chapter[01][1-5].tex
> do
> echo $file
> done
chapter01.tex
chapter02.tex
chapter03.tex
chapter04.tex
chapter05.tex
chapter11.tex
chapter12.tex
```

In the example, the ">" characters are the prompt from the shell showing that the `for` command is not yet finished. Note that the second square bracket uses a dash or hyphen between the 1 and the 5 to include all numbers between 1 and 5. This also works for letters as in `[A-Z]` for all uppercase letters, `[a-z]` for all lowercase letters and `[A-z]` for *all* letters.

Note that the "not" character "`!`" can be used with a range also. For example, "`ls .[!.s-z]`" skips files or directories with two consecutive dots, but it also would skip any file or directory with a name that began with, "`.s`", "`.t`", ... "`.z`". Note that, as with everything else in UNIX, this is case-sensitive. The above expression would not match "`.S`",

or any other combination that included an uppercase character as the second character. All file names with `s-z` as the second character would be displayed.

Now, to introduce what hopefully will only be *minor* confusion. The `grep` (or **g**lobal **r**egular **e**xpression **p**rint) command also uses regular expressions. But, instead of using an exclamation or "bang" as the "not" character, grep and its variations use the caret symbol for negation within the square brackets. Note that this character is also referred to as a "hat" in some usage. But with the greps, outside the square brackets, a caret means the start of the line. A dollar sign "$" means the end of a line. Using `grep` on a file that contains blank lines, we could count the blank lines with the following:

```
$ grep ^$ file | wc -l
10
```

The result is the number of blank lines in the file. A possibly more useful application of this would be removing all blank lines from a file. This could be done with:

```
$ grep -v ^$ file > temporary_file
$ mv temporary_file file
```

Note that the `-v` option for `grep` inverts the meaning of the select expression and outputs the lines that *do not* match the pattern.

Finally, a more complex example, drawn from the lecture notes for *Unix Essentials*. The file `/etc/passwd` contains the mapping between the userid that people use to login to a UNIX system and the numeric uid the the system uses internally to show ownership of files, etc. Additionally, until the mid-1990s, this file also stored the user's encrypted password. (The encrypted password is now in the file `/etc/shadow` for security reasons.)

The `passwd` file has a carefully designed structure that was developed to make it easy to parse. Remember that the early UNIX systems had *extremely* limited RAM by current standards. A machine with 64-128 KBytes would not have been exceptional. So programs had to be small.

The `passwd` file contains the following fields, in order:

userid: what you type when logging in

encrypted password: (actually no longer here - but the field remains)

uid: the "user number" - used in a lot of places in the system

gid: the group number - defined in `/etc/group` and used in a lot of places

GECOS field: the name is historical - it's now used to store real names, office number, telephone extension, etc.

home directory: the full path to the home directory

login shell: the shell to start when the user logs into the system

The fields in the `passwd` file are separated by colons (":"). So, an actual line from a `passwd` file would look like:

```
gingrich:x:556:100:Don Gingrich:/home/g/gingrich:/usr/local/bin/tcsh
```

One problem with using `grep` on a `passwd` or `shadow` file to get the line associated with a particular userid is that there are a lot of short userids on yallara. And grep will find strings that are sub-strings of bigger strings. Using the command "`grep man` *file*" would find lines lines that contain the word "man". But it would also find lines that contain "woman", "Herman", "workman", *etc.* Looking at yallara, the number of three character userids is:

```
egrep "^[a-z][a-z][a-z]:" /etc/passwd | wc -l
     109
```

The above line, which uses `egrep` or *extended* `grep` to allow more complex regexs, counted the number of three letter userids on yallara. It does not require a lot of imagination to realise that many of those userids will also appear as substrings in userids. For that matter simply using `grep` to search for the userid may also find the userid string as a substring in any of the other fields on a line in the file. Simply using "`grep` *userid* `/etc/passwd`" is likely to generate a lot of "false positive" hits in the file. The following script will test this hypothesis:

```
for name in ‘egrep "^[a-z][a-z][a-z]:" /etc/passwd | awk -F: ’{print $1}’‘
do
    echo $name appears on ‘grep $name /etc/passwd | wc -l‘ lines
done
```

The userid bin is the most common, probably since most login shells are located in some sort of "`bin`" directory. In one form or another bin appears on 2400+ lines in the `passwd` file on yallara. Several other userids are on 60-80 lines of the file. Clearly, the above approach to a search string would not work. So, how can a search string be written that correctly selects only the desired line?

```
for name in ‘egrep "^[a-z][a-z][a-z]:" /etc/passwd | awk -F: ’{print $1}’‘
do
    echo $name appears on ‘egrep "^${name}:" /etc/passwd | wc -l‘ line
done
```

Have a look at the book *Mastering Regular Expressions* if you need to take this further.

Regular Expression Operators

Description	Operator	Usage Example	Meaning of	Supported By
Any single Character	`.`	`.in`	Bin, bin, tin, Tin, gin, fin,...	All
Start of line	`^`	`^A`	line that begins with A	All
End of line	`$`	`Z$`	line that ends with Z	All
Escape	`\`	`\$`	matches `$` *only*	All
Character group	`[ ]`	`[abc]`	a *or* b *or* c	All
Negated group	`[^ ]`	`[^abc]`	matches any character that is not a or b or c	All
Grouping	`()` or `\(\)`	`(regexp)`	used in several different ways	All
Optional	`?` `?`	`ac?` `a(df)?`	a *or* ac	awk, egrep
Repetition (0 or more times)	`*`	`qu*`	q, qu, quu,...	All
Repetition (1 or more times)	`+`	`qu+`	qu, quu, quuu,...	All

Table taken from material in *Mastering Regular Expressions* by Jeffrey E. F. Friedl and UNIX*, the Textbook* by Syed M. Sarwar, Robert Koretsky and Syed A. Sarwar.

Managing the Command Line

In general, scripts are written in one of two basic styles. They are either designed to run in the background with little or no direct user interaction, or they are designed to be interactive and to directly respond to input provided by the user as the script is running.

Interactive scripts will typically display a menu and wait for the user to select an option. An interactive script will then go and do the option. Afterwards, the script may either terminate or re-display the menu and wait for a further user response.

A command line script should be designed along the basic UNIX programming conventions. Quoting from Doug M°Ilroy who invented *Unix* pipes, "This is the *Unix* philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface." (Quoted in *A Quarter Century of* UNIX by Peter H. Salus.) Working together is commonly achieved by having programs accept input from `stdin` and sending their normal output to `stdout`. This approach makes it easy to link simple programs together via pipes to accomplish arbitrarily complex tasks.

Command Line Arguments

Those who have programmed in the C language will remember that the full form of the main function is:
`int main(int argc, char **argv)`
The integer, `argc` contains the count of the command line arguments passed to the program, and `argv` is an array of pointers to pointers characters that provides a series of character strings for the arguments. The final element in `argv` is a `NULL` pointer. The first element in `argv` is the name with which the program was called. This can be useful when the same program is called with more than one name and behaves differently depending on the name. An example of this is the `vim` editor. If we use `ls -l` on the directory: `/usr/local/stow/vim-7.0.188/bin` we see the output:

```
total 3168
lrwxrwxrwx   1 root     root           3 Jan 24 16:45 ex -> vim
lrwxrwxrwx   1 root     root           3 Jan 24 16:45 rview -> vim
lrwxrwxrwx   1 root     root           3 Jan 24 16:45 rvim -> vim
lrwxrwxrwx   1 root     root           3 Jan 24 16:45 view -> vim
-rwxr-xr-x   1 root     other    1583424 Jan 24 16:45 vim
lrwxrwxrwx   1 root     root           3 Jan 24 16:45 vimdiff -> vim
-rwxr-xr-x   1 root     other       1600 Jan 24 16:45 vimtutor
-rwxr-xr-x   1 root     other      15772 Jan 24 16:45 xxd
```

The *real* executable file is `vim`. Running the `view` program is equivalent to running `vim` with the `-R` option.

The shell provides access to the same information as is available in the `argc` and `argv`. The shell equivalent of `argc` is `$#` and contains the number of command line arguments that have been passed to the script. The shell equivalent of `argv` is a little messier. `argv[0]` is represented by `$0`. That's fairly simple.

The rest of the arguments are represented by `$1` - `$9`. The obvious question that this raises is, "Does this mean that only nine command line arguments are permitted?" This would be an unreasonable limitation. The solution is the `shift` operator. Calling `shift` within the script moves the value of `$2` to `$1`, the value of `$3` to `$2`, *etc.* and finally the value of the tenth argument becomes `$9`, the eleventh argument becomes the tenth, *etc.*

The variable `$*` contains the entire command line at any stage. Note that the previous value of `$1` is lost when `shift` is called.

Looking at a simple script that demonstrates how command line arguments work:

```
#!/bin/sh

echo "The name of this script is $0"
echo "This script was passed $# command line arguments"
echo "The arguments displayed using \$8 are: $*"
echo "The arguments displayed using \$@ are: $@"

COUNT=1

while [ "$1" != "" ]
do
        echo "Argument number $COUNT is ==>${1}<=="
        shift
        echo "The remaining arguments are: $@"
        COUNT=‘expr $COUNT + 1‘
done
```

The following command line produced the output displayed:

```
> ./test-script foo bar baz zot quux foobar foobaz 1 2 3 4 5
The name of this script is ./test-script
This script was passed 12 command line arguments
The full command line arguments were: foo bar baz zot quux foobar foobaz 1 2 3 4 5
The arguments were foo bar baz zot quux foobar foobaz 1 2 3 4 5
Argument number 1 is ==>foo<==
The remaining arguments are: bar baz zot quux foobar foobaz 1 2 3 4 5
Argument number 2 is ==>bar<==
The remaining arguments are: baz zot quux foobar foobaz 1 2 3 4 5
Argument number 3 is ==>baz<==
The remaining arguments are: zot quux foobar foobaz 1 2 3 4 5
Argument number 4 is ==>zot<==
The remaining arguments are: quux foobar foobaz 1 2 3 4 5
Argument number 5 is ==>quux<==
The remaining arguments are: foobar foobaz 1 2 3 4 5
Argument number 6 is ==>foobar<==
The remaining arguments are: foobaz 1 2 3 4 5
Argument number 7 is ==>foobaz<==
The remaining arguments are: 1 2 3 4 5
Argument number 8 is ==>1<==
```

```
The remaining arguments are: 2 3 4 5
Argument number 9 is ==>2<==
The remaining arguments are: 3 4 5
Argument number 10 is ==>3<==
The remaining arguments are: 4 5
Argument number 11 is ==>4<==
The remaining arguments are: 5
Argument number 12 is ==>5<==
The remaining arguments are:
```

Parsing the Command Line

As a part of the common design of a UNIX command line, the options for a command can be grouped providing that none of the options has a required argument. For example, `ps -efa` could also be written as `ps -e -f -a`. Actually implementing shell code to parse this type of command line and get all of the semantics correct would be difficult. Fortunately, this is not necessary.

There are two programs and a shell built-in function to make this easier. The `getopt` program is an older variant and not as easy to use. It is also being phased out of the Solaris UNIX system. So do not use `getopt`. In any event, for scripts that did use `getopt`, there is a utility, `getoptcvt`, that will convert these scripts to use `getopts` instead.

There is a version of getopts built-in to the Bourne shell and there is a version in `/usr/bin/getopts`. They both have the same options, so, it is probably just as good to use the built-in, as well as being slightly more efficient, it is likely to be simpler since there is no need to define a variable to point to the program name.

The `getopts` command is effectively called as a subroutine or function from your script. It is generally used as the command that controls the behaviour of a `while` loop. The basic call to `getopts` includes the command, a list of the command line options that should be accepted, and the name of a variable. This variable will contain the current option that is being processed. If a letter in the list is followed by the colon '`:`' character, then the letter represents a command line option that requires an argument. An example of this is the `-o` option for the `gcc` compiler. The usage is `-o` *output_file* where is the name of output file that is being created.

A couple of examples of the use of `getopts` taken from the Sun Solaris manual page: The following example script parses and displays its arguments:

```
aflag=
bflag=
while getopts ab: name
do
    case $name in
    a)      aflag=1;;
    b)      bflag=1
        bval="$OPTARG";;
    ?)     printf "Usage: %s: [-a] [-b value] args\n"  $0
        exit 2;;
    esac
done
if [ ! -z "$aflag" ]; then
printf "Option -a specified\n"
fi
if [ ! -z "$bflag" ]; then
    printf 'Option -b "%s" specified\n' "$bval"
fi
shift $(($OPTIND - 1))
printf "Remaining arguments are: %s\n" "$*"
```

Note that the script uses `getopts` in a `while` ‘ loop so that it will continue processing

The following fragment of a shell program processes the arguments for a command that can take the options -a or -b. It also processes the option -o, which requires an option-argument:

```
while getopts abo: c
do
    case $c in
        a | b)   FLAG=$c;;
        o)       OARG=$OPTARG;;
        \?)      echo $USAGE
            exit 2;;
    esac
done
shift ‘expr $OPTIND - 1‘
```

This one is actually a bit less than ideal since, if both `a` and `b` options are specified, whichever is last will be the option that is used. Since the behaviour of a program should be consistent and should not be a surprise to the user, this one really does not qualify.

Functions

As with other programming languages, you can write functions in a shell script. As with most other aspects of programming, shell script functions behave a bit differently in some

respects. The variables passed to a shell script are effectively passed by value, but a function can access any variable in the script since all variables are global within the script.

```
#!/bin/sh
# Script to test Bourne shell functions

testing()
{
    foo=zot
    echo  "variable foo is $foo in function testing"
    return 0
}

testing2()
{
    echo "The variable passed to testing2 was \"$1\""
    return 1
}

foo=bar
echo "variable foo is $foo before function call to testing"
testing
if [ $? -eq 0 ] ; then
    echo testing returned 0
fi
echo "variable foo is $foo after function call to testing"
testing2 "a variable with spaces"
if [ $? -eq 1 ] ; then
    echo testing2 returned 1
fi
```

The output from this script is:

```
$ ./functions
variable foo is bar before function call to testing
variable foo is zot in function testing
testing returned 0
variable foo is zot after function call to testing
The variable passed to testing2 was "a variable with spaces"
testing2 returned 1
```

Scripts for the Background

In the previous section there was a discussion about implementing UNIX standard command line syntax. On of the reasons this can be particularly useful is in scripts that are designed

as "background" jobs. One possible use for this type of script is a multi-step process that takes some considerable amount of computing resources. It would be somewhat impolite to run a compute-intensive series of processes on a shared machine when a lot of others were trying to use the machine. A better option would be to find a way to start it automatically in the middle of the night when the load on the machine is low.

Another problem that can be solved by background jobs running automatically is system maintenance. Some examples of this problem are more common in a university environment, but there are potential problems in any environment that can be solved by scripts that run regularly in the background.

Two examples of this type of problem are the annual problem with Operating Systems students attempting to use the `fork()` function, and the problem several years ago when 200 students all attempted to run 10 instances each of the Apache web server on a machine with 2GB of real RAM.

In both cases the goal of the script was to maintain a stable environment for the majority of students who were doing the "right thing" while removing the threat posed by others who were using excessive resources.

An example of this is the following script that was used a few years back when the Operating Systems assignment involved writing a program called "`node`". The assignment involved a parent process that launched several copies of `node` and sent messages via pipes. But many students' assignments were failing and leaving multiple copies of node on the system. This becomes a critical problem given that process table entries are a limited resource. If there are no available entries, then it is effectively impossible for anyone to login or to run any command, including commands to deal with the problem. The solution was a script to kill the node processes.

The first attempt looked like:

```
#!/bin/sh
ps -efa | grep node | awk '$3 == 1 {print $2}' | xargs kill -9
```

The stages in the pipeline are:

- use the `ps` command to get a list of all processes
- find all processes with node in the command line
- the `awk` command is interesting for two reasons
  - it uses a selection criterion - third field of the `ps` output (the parent process ID -PPID) is 1 which means that the parent has terminated and left the process as an "orphan"
  - it then outputs the process ID of the orphan to the next step in the chain
- finally the `xargs` command sets up a command line for the `kill` command to actually terminate the processes

Note: see the following section for more information about the `xargs` command.

This method *did* remove instances of node from the system. But, it also removed editors that were editing a file named `node.c`. And worse, since it used "`kill -9`" it did not give the editor any chance to save work. So this approach was unacceptable. (Some editors go into a normal state where their parent process is process 1.)

Actually, during the development of the script, the `kill` command was replaced with:
`xargs -n 1 ps -fp`
This calls the `ps` command for each each process that would be deleted by `kill` and lists the full command line. For example, if a user was using the `vim` editor to edit `node.c`, the final field of the `ps` output would be:
`vim node.c`

A quick glance revealed that there were a lot of editor sessions that would be killed. The students would not be happy with this result.

The script was modified to look like this:

```
#!/bin/sh

ps -efa | grep node |\
 grep -v vi |\
 grep -v vim |\
 grep -v pico |\
 grep -v emacs |\
 grep -v nedit |\
 sort |\
 awk '$3 == 1 {print $2}' |\
 xargs -n 1 ps -fp
```

A couple of further comments here. This is an example of a "quick and dirty" admin script. The system was in danger of crashing due to a shortage of available entries in the process table.
It was going to be used for a short time.
Maintainability was not a consideration.
The backslash character at the end of the line is actually a "shell escape" character. It tells the shell to ignore the following character, in this case the "line feed" or newline character that would otherwise terminate the command. Effectively, everything from the first `ps` command to the `xargs` command that sets up the final `ps` command is on *one* command line.

Spreading the line out the way it is makes it easier to exempt another editor if one is found that was missed in the initial writing.

A more complex example is the solution to the problem of multiple HTTP daemons. The actual script no longer exists, unfortunately. The best re-creation of the script looks something like this:

```
#!/bin/sh

AWK=/usr/bin/awk
PS="/usr/bin/ps -ef"
GREP=/usr/bin/grep
WC=/usr/bin/wc
SORT=/usr/bin/sort
XARGS=/usr/bin/xargs
MAILX=/usr/bin/mailx
KILL=/usr/bin/kill
LIMIT=5
MESSAGE="some-file-with-a-message-to-the-user"

# First get a list of users with an httpd instance running

for user in `$PS  | $GREP -v UID | $GREP httpd | $AWK '{print $1}' | $SORT -u -b`
do
    num=`$PS | $GREP $user | $GREP httpd | $WC -l` # number of daemons running
        if [ $num -gt $LIMIT ] ; then
        $PS | $GREP $user | $GREP httpd | $AWK '{print $2}' | $XARGS $KILL -9
            $MAILX -s "Apache Daemons killed" ${user}@cs.rmit.edu.au < $MESSAGE
    fi
done
```

OK, what happened here? The long string of plumbing (stuff connected by pipes) in the for statement is designed to generate a list of users who have at least one instance of the Apache (`httpd`) daemon running. Killing all of the daemons would be relatively easy, but would penalise students who did the right thing and limited the the number of daemons in their account. But there is no simple single line way to test the number of instances of daemon for each user. The line that begins with "`num=` sets the variable `num` equal to the number of `httpd` instances for `$user`. If the number of daemons is greater than the LIMIT, then the daemons will be killed. The first line after the `if` gets the process IDs of all processes in the system, the first `grep` extracts the processes owned by the user, the next `grep` extracts the instances of the `httpd` daemon, the `awk` grabs the process ID for these processes and passes them, via `xargs` to `kill` to remove them. Note that there were a *lot* of `httpd` instances, so limiting on the userid first has a processing advantage.

If the system was in a critical state and it was necessary to kill *all* Apache daemons, the following line would do the job:

```
ps -efa | grep httpd |awk 'print $2' | xargs kill -9
```

But, as was noted above, this would not have been fair to the students who had edited their Apache consideration to limit the maximum number of daemons.

You would also note that the script correctly begins with variable definitions for all of the programs used. This script was designed to run as a `cron` job. One of the characteristics of `cron` is that it limits or removes the path that is available to commands in their environment. If a script-writer wants to make sure that the correct versions of commands are used, the best way to do this is to use the variable as illustrated.

A short note about `cron`

The `cron` command and the crontab command are designed to allow a user, and more often a system administrator, to run commands in the background at regular intervals. It is another of the commands that has been in UNIX for a long time. And, because it has been around for a while, it has a fairly unforgiving syntax. But the power of being able to schedule repetitive tasks so that they happen automatically makes up for the painful syntax.

The manual pages for `cron` and `crontab` ar more readable than most manual pages. It is worth taking the time to read them.

There is also a command named `at` that can be used to run another command *once* at some particular time in the future. It also has fairly painful syntax, but is useful enough to learn about.

Commands Useful in Scripts

As you should have noticed if you have read this far, most scripts tend to string together a variety of UNIX commands, connecting them with pipes and IO redirection, storing intermediate results in temporary files, and finally achieving their intended purpose.

This section is going to look at several UNIX commands that are frequently used in scripts and how these commands are commonly used. For more detail about these commands, se the relevant manual pages, or if the discussion mentions available texts, try to get one of these and read it.

`expr`

The `expr` command is designed to do mathematics in a shell script. It can accept shell variables as arguments. Unfortunately, `expr` only does integer maths. This is a disadvantage, but there are still lots of things that can be usefully done with integers.

This command was used in several example scripts incrementing loop counters.

The `xargs` command is one of the most powerful commands availableto a scriptwriter. There are lots of times when the output from a pipeline is some list of values that need to be processed as the command line arguments (rather than `stdin`) for some other command. `xargs` is the tool to do what is required. In its normal form, "`xargs` *`somecommand`*", `xargs` will place the command on a command line and grab input from `stdin` inserting it into the command line untill either an "end-of-file" character is encountered or the command line

is bigger than is allowed under the particular version of UNIX. When the limit is reached, `xargs` will `fork()` and one of the "`exec()`" functions to run the command. If running the command was not triggered by an end-of-file, `xargs` will wait for more input and continue.

The options available for `xargs` can be fairly complex since the command has a variety of wayd of doing things. Probably the most common single option is the "`-n 1`" option which means that a new instance of the program will be launched for each new argument that comes down the pipeline. This is useful for commands that do not allow multiple targets on the command line.

`test`

The `test` command is used in a lot of places in scripts, but most scripts never actually have the command name "`test` in them. The reason for this is that a "right square bracket" '`[`' is equivalent to the command `test`.

So, for example, in an `if` statement, the statement could actually be written in one of two equivalent ways.

```
if [ $FOO = "a" ]
    then
    echo "FOO is a"
fi
```

or

```
    if test $FOO = "a"
    then
    echo "FOO is a"
fi
```

The square bracket usage just makes it easier for programmers to see what is happening.

14cm **Note**, you could put any program or series of programs in an `if` statement. The return code from the last program in the string will be the switch for the `if` statement. But remember that true/OK is 0 in UNIX and non-0 is false/not-OK.

Just to demonstrate that point:

```
$ if echo foo
> then
> echo "true"
> fi
foo
true
```

In the demonstration, the commands are entered directly at a Bourne shell prompt. For what it is worth, the `echo` command has a return code of 0 in almost all circumstances.

The `test` command has a wide variety of comparison and test operators. This discussion will only touch on some of the more commonly used variations.

Because test was one of the commands in the early UNIX systems, it has a few quirks, unfortunately. The primary point in the interface that is likely to be confusing to a new user is the way comparisons are handled. Logically, it would seem to make sense that the comparison operators, `=`, >, <, etc. would be numeric comparisons since the symbols are the common mathematical symbols. But, these symbols are used for string comparisons similar to the comparisons in the `strcmp()` 'C' language functions. Test returns a value of zero if the strings are the same, and non-zero if they are different.

If a numeric comparison is required, the equivalent comparison operators for numeric comparisons include, `-eq`, `-gt`, and `-lt`, etc. See the manual page for test for more details, and for comparisons such as "greater than or equal", etc.

The numeric comparisons use what logically look like string comparison operators because the mathematical symbols were already in use for strings. And, while it might have made sense to sort this out, changing the operators' behaviour would have broken existing scripts. With UNIX, as with most operating systems, there is a strong motivation to maintain backward compatibility.

Relative and Absolute Paths

The operand for the next groups of `test` operators is the *name* of a file or directory. These names can be either relative or absolute filenames. An absolute or "fully-qualified" file or directory name is one that begins at the root of the filesystem and traces the directories and sub-directories to the file. An example of this would be "`/usr/bin/ls`". A relative pathname is effectively a "map" to the file or directory from the current working directory. Examples of this would include:
`foo` - the file "`foo`" is in the current directory.
`./foo` - another way of saying, "The file `foo` in the current directory."
`../foo` - `foo` is in the parent of the current directory.
`bar/foo` - `foo` is in a directory named `bar` which is a sub-directory of the current directory.
Finally, a complex example of a relative pathname.
`../bar/foo` - `foo` is in a directory named `bar` that has the same parent directory as the current directory.
Note that relative pathnames can include multiple "`../` groups so long as the string of parents does not attempt to go below the root of the filesystem. This type of complex relative pathname is sometimes found in symbolic links that are designed to be equivalent no matter how the filesystem is mounted.

Filesystem `test` Operators

The next group of test operators are those used on the filesystem. Some of the more commonly used operators in this group include the following. The `-f` operator is true if the operand is the name of a normal file. The `-d` operator is true if the operand is the name of a directory. And the `-s` operator returns true if the operand is the name of a file that has a length greater than zero. Using the `touch` command can create a file with a zero length. The operand can be either a relative or absolute pathname.

Finally, there are operators related to the filesystem security in UNIX. The operators,

`-r`, `-w`, and `-x` relate to the permission bits on the operator, which can be either a file, a directory, or any other object in the filesystem. The permission in question is whether the script, running with its current permissions, is permitted the particular type of access.

So, for example, if the script is run by the `root` user on a system, and the test is "`-r`" then the return code will always be "true" or zero. This is because `root` can read any file in the system The `root` user also has write access to any file, so `-w` will always be true for `root`.

Whether `-x` returns true or false will depend on whether any of the "execute" bits are set. This is true even for `root`.

This represents an overview of the available options. See the manual page for `test` for the complete list of available tests.

`find`

The `find` command can be used to search the filesystem. It can search for files on a variety of conditions. For example, find can find files that have not been accessed for over a year. This could be useful in designing a script to manage archiving of files.

It can search for regular files or directories. Combining `find` with `chmod` a script could be written to change the permissions on a directory, its subdirectories, and the files in them. Clearly if the directories were to be accessible, there is a need for different permissions than for the files.

The general form of the command is:
`find` *start_directory options operator*
Where:

start_directory - where to start the search. This can be a dot ' . ' for the current directory. A relative path or an absolute path.

options - one or more selection criteria. These will be discussed in the table below.

operator - what to do with files or directories that meet the selection criteria.

`find` Arguments and Options		
Option	Arguments	What it does
-d	none	Default is a "breadth first" search – modifies this to depth first
`-atime`	time	I-nodes contain access time (`atime`)
`-ctime`	in	time the I-node was last modified (`ctime`)
`-mtime`	days	time the file contents were last modified `mtime`
Time Arguments		
`+`n		more than n days
n		exactly n days
`-`n		less than n days
`-size` n	size	Checks the size of the file in blocks (Blocks are about 512 bytes)
`-size` n`c`	size	Checks the size of the file in bytes
Size Arguments		

	`n`
	`+n`
	`-n`
		exactly n bytes or blocks
		greater than n bytes or blocks
		less than n bytes or blocks

`-name`	regular expression	Searches for files by name or by globbing expression - a globbing expression *must* be in quotes so the shell does not try to expand it.

`-type`	
	`f`
	`d`
	`l`
		Find files, directories or other filesystem contents
		regular file
		directory
		symbolic links

`-perm`	mode -mode	exactly the specified permission bits "set" true if the specified bits are "set" even if other bits are also set. mode is an octal number
`-nouser`		the uid in the I-node does not appear in the `/etc/passwd` file

`file`

The `file` command makes an attempt to determine the contents of a file. Since UNIX does not use the Windows approach of defining file contents by means of a filename extension, there is a need for an easy way to verify file contents.

Usage is `file` *filename*

The command will report the type of contents of the file for a fair range of contents. The file `/etc/magic` contains the list of filetypes that the `file` command can identify. This includes the ELF format used for executable files, PDF files, various font files, source code for a variety of programming languages, core files, *etc.*

sed

sed is the **s**tream **ed**itor. In general, most of the command mode options from the vi or vim editor will also work in sed

A common use of sed is in renaming files with some specific pattern. Another use is a global search and replace on a list of files in a batch.

sed Arguments		
Option	Arguments	What it does
s	/*find*/*replace*/	substitute replace text for find regexp
y	/*string1*/*string2*/	substitute characters in string two for characters in string one
d	/*pattern*/d	delete lines containing pattern

Note that the substitute regexp can contain parenthesis, which, if escaped as in \(\) allow the text from the find to be inserted into the replace text. As, for example the following:

```
s/^\([^ ]*\).*$/\1/
```

will grab the first "word" of each line of a file, where word is defined as non-space characters, terminated by a space. Another note is that to avoid complex sed expressions with multiple "escaped" slashes. (\/), the / character can be replaced with an alternate punctuation character that is not used in the search or replace string. The colon ':' character is commonly used for this purpose.

A common mistake that is made using sed is to attempt to send the output to the source file with an I/O redirect. The problem with this is that the > deletes the contents of the file before the command is run. Thus the source disappears before it can be processed. The solution is the following construct:

sed -s/*pattern*/*pattern*/ *source* > temp${$} && mv temp${$} *source*

The effect of the && is to make sure that the mv command only runs if the sed was successful.

awk

awk (named for Aho, Weinberger, and Kernigan) is a complete programming language in its own right. It is possible to write large and complex scripts in awk. **But don't hand in an awk script for an assignment that specifies a shell script.**

The biggest strength of awk is its ability to handle lines of input and break it into fields.

A particularly useful awk option is the -F option which allows a user to specify a field separator. (see the example script that parsed a passwd file)

In addition, there are operators that allow formatted output of the fields from a line and comparison operators so that only lines where a particular field meets some criterion will be acted upon.

`sed` and `awk` are also the topic of a book from O'Reilly. Clearly they represent another fairly large topic.

Running Commands

A program can be run by simply typing its name if it:

- Is either a script or an executable binary filetype
- Has the execute permission bit set to allow the user who has typed the command to actually run it.
- Is in one of the directories that are in the path that is part of the user's environment

You should note that the first instance of a file of the name that was typed will be the one that runs.

```
gingrich@goanna 281> ls -l /usr/ucb/shutdown
-rwxr-xr-x 1 root bin 17024 Jan 6 2000 /usr/ucb/shutdown
gingrich@goanna 282> ls -l /usr/sbin/shutdown
-rwxr-xr-x 1 root sys 4094 Jan 6 2000 /usr/sbin/shutdown
gingrich@goanna 283> echo $path
/home/g/gingrich/bin /usr/local/bin /usr/local/sbin /usr/local/bin/X11 /bin
/usr/bin /sbin /usr/sbin /usr/local/pkg/teTeX/bin /usr/local/pkg/netpbm /opt/SUNWspro
/usr/ccs/bin /usr/ucb
```

So, in the above example, if I was logged in as root with the path set as displayed, which of the two clearly different `shutdown` commands would run if I typed `shutdown`?

Following down the path it should be clear that `/usr/sbin/shutdown` is the version that would run.

Two other useful commands in this context are the `whereis` and `which` commands.

```
gingrich@goanna 285> whereis shutdown
shutdown:  /etc/shutdown /usr/sbin/shutdown /usr/ucb/shutdown
gingrich@goanna 286> which shutdown
/usr/sbin/shutdown
```

The `whereis` command has a predefined path that includes a number of directories where commands are often found. On some Linux systems `whereis` also looks in the `man` page directories also. On the other hand, `which` follows the current path and will report the first command of the given name that is found. This represents the version that would run

Use of the Semicolon and Parenthesis in Scripting and Command Lines

The semicolon `;` is often used in both scripts and command lines as a substitute for a newline. A newline is the usual way to separate commands. A semicolon is equivalent.

Parentheses `( )` are a way of grouping commands together.

As an example consider the following line. The assumption is that we are in some directory that contains a number of files and sub-directories. The task is to move them to `/newdirectory` .

```
# tar cvpf - * | ( cd /newdirectory ; tar xf - )
```

What does this do? The initial tar command will grab all of the files and directories in the current directory, preserving the ownership and permissions, and send them to `stdout` The pipe symbol `|` redirects `stdout` to `stdin` of the following command(s). The `cd` command does not do anything with `stdin` so it gets passed to the `tar` command that extracts the files into the `/newdirectory` .

One Last Look at the Path

In the section where we were running scripts we needed to type `./`*scriptname* to run the script since the directory where the script is located is not in the current path. So, would it be a good idea to put the current directory in the path by adding a dot "." to the path? The answer is **No!**

The reason for this is that, when you are running as root, having the current directory in your path is dangerous.

Consider the following devious little script:

```
#!/bin/sh
cp /bin/sh .
chown root:root ./sh
chmod 4555 ./sh
ls ${*}
```

If this script is put in the current directory and root has a dot in the path near the start then this will create a shell command that the user will be able to use later and gain root access to the system.

The devious user could create a file with a weird name and ask the administrator for assistance in removing it, to give just one example of how this could happen.

Note that creating scripts in a personal `bin` directory that *is* in the path is a good idea and will eliminate the need for the leading `./` There is one small caveat, though. Some shells build lists of the executable file names the first time that they search the path. If you create a new script in the `$HOME/bin` directory, it will not be runnable directly until you use the `rehash` shell directive. In particular this is true for the `tcsh`.