Bash-Scripting

An introduction

Script for the Course UNIX II

Dipl. Inf. Heiko Schellhorn

Institute of Environmental Physics

Bremen University

Table of Content

Introduction 3

  Purpose Of This Tutorial 3

  A Brief History of sh 3

  Audience 3

  Typographical Conventions Used in This Tutorial 3

Philosophy 5

A First Script 7

Variables - Part I 9

  Scope of Variables 10

Wildcards 13

Escape Characters 14

Loops 16

  For Loops 16

  While Loops 17

Test 20

Case 24

Variables - Part II 26

External Programs 29

Functions 30

Scope of Variables 31

Recursion 32

Exit Codes 34

Anhang 35

  Exit Codes 35

Introduction

Purpose Of This Tutorial

This tutorial is written for MScEP students to understand some of the basics of shell script programming, and hopefully to introduce some of the possibilities of simple but powerful programming available under the bourne shell. As such, it has been written as a basis for one-on-one or group tutorials and exercises, and as a reference for subsequent use.
Whilst this tutorial was written for MScEP-Students in mind, it is publicly available as a useful resource for all. This is partly as an example of the Free Software philosophy - although this document was created for internal use, once it has been written, there is no additional cost involved in making it available to all.

A Brief History of sh

Steve Bourne, wrote the Bourne shell which appeared in the Seventh Edition Bell Labs Research version of Unix.

Many other shells have been written; this particular tutorial concentrates on the Bourne and the Bourne Again shells.

Other shells include the Korn Shell (ksh), the C Shell (csh), and variations such as tcsh. This tutorial does *not* cover those shells. Maybe a future version will cover ksh; I do not intend to write a tutorial for csh, as csh programming is considered harmful.

Audience

This tutorial assumes some prior experience; namely:

- Use of an interactive Unix shell
- Minimal programming knowledge - use of variables, functions, is useful background knowledge
- Understanding of how Unix commands are structured, and competence in using some of the more common ones.
- Programmers of C, Pascal, or any programming language who can maybe read shell scripts, but don't feel they understand exactly how they work.

Typographical Conventions Used in This Tutorial

Significant words will be written in *italics* when mentioned for the first time.

Code segments and script output will be displayed as preformatted text.
Command-line entries will be preceded by the Dollar sign ($). If your prompt is different, enter the command:

```
PS1="$ " ; export PS1
```

Then your interactions should match the examples given (such as $ ./my-script.sh below).
Script output (such as "Hello World" below) is displayed at the start of the line.

```
$ echo "#!/bin/sh" > my-script.sh
$ echo "echo Hello World" >> my-script.sh
$ chmod 755 my-script.sh
$ ./my-script.sh
Hello World
$
```

Entire scripts will be surrounded by thick horizontal rules and include a reference where available to the plain text of the script:

first.sh

```
#!/bin/sh
# This is a comment!
echo Hello World        # This is a comment, too!
```

Note that to make a file executable, you must set the eXecutable bit, and for a shell script, the Readable bit must also be set:

```
$ chmod a+rx first.sh
```

Philosophy

Shell script programming has a bit of a bad press amongst some Unix systems administrators. This is normally because of one of two things:

- The speed at which an interpreted program will run as compared to a C program, or even an interpreted Perl program.
- Since it is easy to write a simple batch-job type shell script, there are a lot of poor quality shell scripts around.

It is partly due to this that there is a certain machismo associated with creating *good* shell scripts. Scripts which can be used as CGI programs, for example, without losing out too much in speed to Perl (though both would lose to C, in many cases, were speed the only criteria).
There are a number of factors which can go into good, clean, quick, shell scripts.

- The most important criteria must be a clear, readable layout.
- Second is avoiding unnecessary commands.

A clear layout makes the difference between a shell script appearing as "black magic" and one which is easily maintained and understood.
You may be forgiven for thinking that with a simple script, this is not too significant a problem, but two things here are worth bearing in mind.

1. First, a simple script will, more often than anticipated, grow into a large, complex one.
2. Secondly, if nobody else can understand how it works, you will be lumbered with maintaining it yourself for the rest of your life!

Something about shell scripts seems to make them particularly likely to be badly indented, and since the main control structures are if/then/else and loops, indentation is critical for understanding what a script does.

One of the major weaknesses in many shell scripts is lines such as:

```
cat /tmp/myfile | grep "mystring"
```

which would run much faster as:

```
grep "mystring" /tmp/myfile
```

Not much, you may consider; the OS has to load up the `/bin/grep` executable, which is a reasonably small 75600 bytes on my system, open a `pipe` in memory for the transfer, load and run the `/bin/cat` executable, which is an even smaller 9528 bytes on my system, attach it to the input of the pipe, and let it run.

Of course, this kind of thing is what the OS is there for, and it's normally pretty efficient at doing it. But if this command were in a loop being run many times over, the saving of not locating and loading the `cat` executable, setting up and releasing the pipe, can make some difference, especially in, say, a CGI environment where there are enough other factors to slow things down without the script itself being too much of a hurdle.
Some Unices are more efficient than others at what they call "building up and tearing down processes" - ie, loading them up, executing them, and clearing them away again. But however good your flavour of Unix is at doing this, it'd rather not have to do it at all.

As a result of this, you may hear mention of the Useless Use of Cat Award (UUoC), also known in some circles as **The Award For The Most Gratuitous Use Of The Word Cat In A Serious Shell Script** being bandied about on the `comp.unix.shell` newsgroup from time to time. This is purely a way of peers keeping each other in check, and making sure that things are done right.

Speaking of which, I would like to reccommend the `comp.os.unix.shell` newsgroup to you, although its signal to noise ratio seems to have decreased in recent years. There are still some real gurus who hang out there with good advice for those of us who need to know more (and that's all of us!). Sharing experiences is the key to all of this - the reason behind this tutorial itself, and we can all learn from and contribute to open discussions about such issues.

Which leads me nicely on to something else: Don't *ever* feel too close to your own shell scripts; by their nature, the source cannot be closed. If you supply a customer with a shell script, s/he can inspect it quite easily. So you might as well accept that it will be inspected by anyone you pass it to; use this to your advantage with the GPL - encourage people to give you feedback and bugfixes for free!

A First Script

For our first shell script, we'll just write a script which says "Hello World". We will then try to get more out of a Hello World program than any other tutorial you've ever read :-)
Create a file (first.sh) as follows:

first.sh

```
#!/bin/sh
# This is a comment!
echo Hello World        # This is a comment, too!
```

The first line tells Unix that the file is to be executed by /bin/sh. This is the standard location of the Bourne shell on just about every Unix system. If you're using GNU/Linux, /bin/sh is normally a symbolic link to bash.

The second line begins with a special symbol: #. This marks the line as a comment, and it is ignored completely by the shell.

The only exception is when the *very first* line of the file starts with `#!` - as ours does. This is a special directive which Unix treats specially. It means that even if you are using csh, ksh, or anything else as your interactive shell, that what follows should be interpreted by the Bourne shell.

Similarly, a Perl script may start with the line `#!/usr/bin/perl` to tell your interactive shell that the program which follows should be executed by perl. For Bourne shell programming, we shall stick to `#!/bin/sh`.

The third line runs a command: `echo`, with two parameters, or arguments - the first is "Hello"; the second is "World".
Note that echo will automatically put a single space between its parameters.

The # symbol still marks a comment; the # and anything following it is ignored by the shell.

Now run `chmod 755 first.sh` to make the text file executable, and run `./first.sh`.
Your screen should then look like this:

```
$ chmod 755 first.sh
$ ./first.sh
Hello World
$
```

You will probably have expected that! You could even just run:

```
$ echo Hello World
Hello World
$
```

Now let's make a few changes.

First, note that `echo` puts ONE space between its parameters. Put a few spaces between "Hello" and "World". What do you expect the output to be?
What about putting a TAB character between them?

As always with shell programming, try it and see.

The output is exactly the same! We are calling the echo program with two arguments; it doesn't care any more than cp does about the gaps in between them.

Now modify the code again:

first.sh modified

```
#!/bin/sh
# This is a comment!
echo „Hello    World“        # This is a comment, too!
```

This time it works. You probably expected that, too, if you have experience of other programming languages. But the key to understanding what is going on with more complex command and shell script, is to understand and be able to explain: WHY?

echo has now been called with just ONE argument - the string "Hello   World". It prints this out exactly.
The point to understand here is that the shell parses the arguments BEFORE passing them on to the program being called. In this case, it strips the quotes but passes the string as one argument.

As a final example, type in the following script. Try to predict the outcome before you run it:

first2.sh

```
#!/bin/sh
# This is a comment!
echo "Hello      World"       # This is a comment, too!
echo "Hello World"
echo "Hello * World"
echo Hello * World
echo Hello      World
echo "Hello" World
echo Hello "     " World
echo "Hello \"*\" World"
echo `hello` world
echo 'hello' world
```

Is everything as you expected? If not, don't worry! These are just some of the things we will be covering in this tutorial ... and yes, we will be using more powerful commands than echo!

Variables - Part I

Just about every programming language in existance has the concept of *variables* - a symbolic name for a chunk of memory to which we can assign values, read and manipulate its contents.
The bourne shell is no exception, and this section introduces idea. This is taken further in Variables Part II which looks into variables which are set for us by the environment.

Let's look back at our first Hello World example. This could be done using variables (though it's such a simple example that it doesn't really warrant it!)

Enter the following code into var1.sh:

var.sh

```
#!/bin/sh
MY_MESSAGE="Hello World"
echo $MY_MESSAGE
```

This assigns the string "Hello World" to the variable MY_MESSAGE then echoes out the value of the variable.

Note that we need the quotes around the string Hello World. Whereas we could get away with `echo Hello World` because echo will take any number of parameters, a variable can only hold one value, so a string with spaces must be quoted to that the shell knows to treat it all as one. Otherwise, the shell will try to execute the command World after assigning MY_MESSAGE=Hello

The shell does not care about types of variables; they may store strings, integers, real numbers - anything you like.
People used to Perl may be quite happy with this; if you've grown up with C, Pascal, or worse yet Ada, this may seem quite strange.
In truth, these are all stored as strings, but routines which expect a number can treat them as such.
If you assign a string to a variable then try to add 1 to it, you will not get away with it:

```
$ x="hello"
$ y=`expr $x + 1`
expr: non-numeric argument
$
```

Since the external program `expr` only expects numbers. But there is no syntactic difference between:

```
MY_MESSAGE="Hello World"
MY_SHORT_MESSAGE=hi
MY_NUMBER=1
MY_PI=3.142
MY_OTHER_PI="3.142"
MY_MIXED=123abc
```

Note though that special characters must be properly escaped to avoid interpretation by the shell. This is discussed further in Escape Characters.

We can interactively set variable names using the `read` command; the following script asks you for your name then greets you personally:

```
var2.sh

#!/bin/sh
echo What is your name?
read MY_NAME
echo "Hello $MY_NAME - hope you're well."
```

A reader kindly pointed out to me that I had originally missed out the double-quotes in line 3, which meant that the single-quote in the word "you're" was unmatched, causing an error. It is this kind of thing which can drive a shell programmer crazy, so watch out for them!

This is using the shell-builtin command read which reads a line from standard input into the variable supplied.
Note that even if you give it your full name and don't use double quotes around the echo command, it still outputs correctly. How is this done? With the MY_MESSAGE variable earlier we had to put double quotes around it to set it.
What happens, is that the read command automatically places quotes around its input, so that spaces are treated correctly.

Scope of Variables

Variables in the bourne shell do not have to be declared, as they do in languages like C.
But if you try to read an undeclared variable, the result is the empty string. You get no warnings or errors.
This can cause some subtle bugs - if you assign MY_OBFUSCATED_VARIABLE=Hello and then echo $MY_OSFUCATED_VARIABLE, you will get nothing (as the second OBFUSCATED is mis-spelled).

There is a command called export which has a fundamental effect on the scope of variables. In order to really know what's going on with your variables, you will need to understand something about how this is used.

Create a small shell script, myvar2.sh:

```
myvar2.sh

#!/bin/sh
echo "MYVAR is: $MYVAR"
MYVAR="hi there"
echo "MYVAR is: $MYVAR"
```

Now run the script:

```
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

MYVAR hasn't been set to any value, so it's blank. Then we give it a value, and it has the expected result.

Now run:

```
$ MYVAR=hello
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

It's still not been set! What's going on?!

When you call myvar2.sh from your interactive shell, a new shell is spawned to run the script. This is partly because of the #!/bin/sh line at the start of the script, which we discussed earlier. We need to export the variable for it to be inherited by another program - including a shell script.

Type:

```
$ export MYVAR
$ ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
```

Now look at line 3 of the script: this is changing the value of MYVAR. But there is no way that this will be passed back to your interactive shell. Try reading the value of MYVAR:

```
$ echo $MYVAR
hello
$
```

Once the shell script exits, its environment is destroyed. But MYVAR keeps its value of hello within your interactive shell.

In order to receive environment changes back from the script, we must *source* the script - this effectively runs the script within our own interactive shell, instead of spawning another shell to run it.

We can source a script via the "." command:

```
$ MYVAR=hello
$ export MYVAR
$ echo $MYVAR
hello
$ . ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
$ echo $MYVAR
hi there
```

The change has now made it out into our shell again! This is how your .profile or .bash_profile file works, for example.

One other thing worth mentioning at this point about variables, is to consider the following shell script:

```
user.sh

#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called $USER_NAME_file"
touch $USER_NAME_file
```

Think about what result you would expect. For example, if you enter "steve" as your USER_NAME, should the script create steve_file?

Actually, no. This will cause an error unless there is a variable called USER_NAME_file. The shell does not know where the variable ends and the rest starts. How can we define this? The answer is, that we enclose the variable itself in curly brackets:

```
user.sh        modified

#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called ${USER_NAME}_file"
touch ${USER_NAME}_file
```

The shell now knows that we are referring to the variable USER_NAME and that we want it suffixed with _file. This can be the downfall of many a new shell script programmer, as the source of the problem can be difficult to track down.

Also note the quotes around "${USER_NAME}_file" - if the user entered "Steve Parker" (note the space) then without the quotes, the arguments passed to touch would be Steve and Parker_file - that is, we'd effectively be saying touch Steve Parker_file, which is two files to be touched, not one. The quotes avoid this.

Wildcards

Wildcards are really nothing new if you have used Unix at all before.

It is not necessarily obvious how they are useful in shell scripts though.
This section is really just to get the old grey cells thinking how things look when you're in a shell script - predicting what the effect of using different syntaxes are. This will be used later on, particularly in the Loops section.

Think first how you would copy all the files from /tmp/a into /tmp/b. All the .txt files? All the .html files?

Hopefully you will have come up with:

```
$ cp /tmp/a/* /tmp/b/
$ cp /tmp/a/*.txt /tmp/b/
$ cp /tmp/a/*.html /tmp/b/
```

Now how would you list the files in /tmp/a/ without using ls /tmp/a/ ?

How about echo /tmp/a/*? What are the two key differences between this and the ls output?

How can this be useful? Or a hinderance?

How could you rename all .txt files to .bak?

Note that

```
$ mv *.txt *.bak
```

will not have the desired effect; think about how this gets expanded by the shell before it is passed to mv. Try this using echo instead of mv if this helps.

We will look into this further later on, as it uses a few concepts not yet covered.

Escape Characters

Certain characters are significant to the shell; we have seen, for example, that the use of double quotes (") characters affect how spaces and TAB characters are treated, for example:

```
$ echo Hello       World
Hello World

$ echo "Hello       World"
Hello     World
```

So how do we display: Hello   "World" ?

```
$ echo "Hello   \"World\""
```

The first and last " characters wrap the whole lot into one parameter passed to `echo` so that the spacing between the two words is kept as is. But the code:

```
$ echo "Hello   "World""
```

would be interpreted as three parameters:

- "Hello   "
- World
- ""

So the output would be

```
Hello    World
```

Note that we lose the quotes entirely. This is because the first and second quotes mark off the Hello and following spaces; the second argument is an unquoted "World" and the third argument is the empty string; "".

Most characters (`*`, `'`, etc) are not interpreted (ie, they are taken literally) by means of placing them in double quotes ("). They are taken as is and passed on the the command being called. An example using the asterisk (*) goes:

```
$ echo *
case.html escape.html first.html functions.html hints.html index.html ip-primer.txt raid1+0.txt

$ echo *txt
ip-primer.txt raid1+0.txt

$ echo "*"
*

$ echo "*txt"
*txt
```

In the first example, * is expanded to mean all files in the current directory.
In the second example, *txt means all files ending in `txt`.
In the third, we put the * in double quotes, and it is interpreted literally.
In the fourth example, the same applies, but we have appended `txt` to the string.

However, `"`, `$`, `` ` ``, and `\` are still interpreted by the shell, even when they're in double quotes. The backslash (\) character is used to mark these special characters so that they are not interpreted by the shell, but passed on to the command being run (for example, `echo`).

So to output the string: (Assuming that the value of $MY_VAR is 5):

```
A quote is ", backslash is \, backtick is `, a few spaces are    and dollar is $. $MY_VAR is 5.
```

we would have to write:

```
$ echo "A quote is \", backslash is \\, backtick is \`, a few spaces are    and dollar is \$. \
$MY_VAR is ${MY_VAR}."
A quote is ", backslash is \, backtick is `, a few spaces are    and dollar is $. $MY_VAR is 5.
```

We have seen why the " is special for preserving spacing. Dollar is special because it marks a variable, so $MY_VAR is replaced by the shell with the contents of the variable MY_VAR. Backslash is special because it is itself used to mark other characters off; we need the following options for a complete shell:

```
$ echo "This is \\ a backslash"
This is \ a backslash

$ echo "This is \" a quote and this is \\ a backslash"
This is " a quote and this is \ a backslash
```

So backslash itself must be escaped to show that it is to be taken literally. The other special character, the backtick, is discussed later in External Programs .

Loops

Most languages have the concept of loops: If we want to repeat a task twenty times, we don't want to have to type in the code twenty times, with maybe a slight change each time.

As a result, we have FOR and WHILE loops in the Bourne shell. This is somewhat fewer features than other languages, but nobody claimed that shell programming has the power of C.

For Loops

for loops iterate through a set of values until the list is exhausted:

for.sh

```
#!/bin/sh
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done
```

Try this code and see what it does. Note that the values can be anything at all:

for2.sh

```
#!/bin/sh
for i in hello 1 * 2 goodbye
do
  echo "Looping ... i is set to $i"
done
```

is well worth trying.

Make sure that you understand what is happening here. Try it without the * and grasp the idea, then re-read the Wildcards section and try it again with the * in place.
Try it also in different directories, and with the * surrounded by double quotes, and try it preceded by a backslash (*)

While Loops

while loops can be much more fun! (depending on your idea of fun, and how often you get out of the house...)

while.sh

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
  echo "Please type something in (bye to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

What happens here, is that the echo and read statements will run indefinitely until you type "bye" when prompted.

Review Variables – Part I to see why we set INPUT_STRING=hello before testing it. This makes it a repeat loop, not a traditional while loop.

The colon (:) always evaluates to true; whilst using this can be necessary sometimes, it is often preferrable to use a real exit condition. Compare quitting the above loop with the one below; see which is the more elegant. Also think of some situations in which each one would be more useful than the other:

while2.sh

```
#!/bin/sh
while :
do
  echo "Please type something in (^C to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

Another useful trick is the `while read f` loop. This example uses the case statement, which we'll cover later. It reads from the file myfile, and for each line, tells you what language it thinks is being used.

while3a.sh

```
#!/bin/sh
while read f
do
  case $f in
        hello)          echo English    ;;
        howdy)          echo American   ;;
        gday)           echo Australian ;;
        bonjour)        echo French     ;;
        "guten tag")    echo German     ;;
        *)              echo Unknown Language: $f ;;
   esac
done < myfile
```

On many Unix systems, this can be also be done as

while3b.sh

```
#!/bin/sh
while f=`line`
do
  .. process f ..
done < myfile
```

But since the `while read f` works with any *nix, and doesn't depend on the external program `line`, the former is preferable. See External Programs to see why this method uses the backtick (`).

Thanks again to *a user* for pointing out that I referred to `$i` in the default ("Unknown Language") case above - you will get no warnings or errors in this case, even though `$i` has not been declared or defined. For example:

```
$ i=THIS_IS_A_BUG

$ export I

$ ./while3.sh something
Unknown Language: THIS_IS_A_BUG
```

So make sure that you avoid typos. This is also another good reason for using `${x}` and not just `$x` - if `x="A"` and you want to say "A1", you need `echo ${x}1`, as `echo $x1` will try to use the variable `x1`, which may not exist, or may be set to B2.

A handy Bash (but not Bourne Shell) tip I learned recently from the Linux From Scratch project is:

```
mkdir rc{0,1,2,3,4,5,6,S}.d
```

instead of the more cumbersome:

```
for runlevel in 0 1 2 3 4 5 6 S
do
  mkdir rc${runlevel}.d
done
```

And this can be done recursively, too:

```
$ cd /
$ ls -ld {,usr,usr/local}/{bin,sbin,lib}
drwxr-xr-x    2 root     root         4096 Oct 26 01:00 /bin
drwxr-xr-x    6 root     root         4096 Jan 16 17:09 /lib
drwxr-xr-x    2 root     root         4096 Oct 27 00:02 /sbin
drwxr-xr-x    2 root     root        40960 Jan 16 19:35 usr/bin
drwxr-xr-x   83 root     root        49152 Jan 16 17:23 usr/lib
drwxr-xr-x    2 root     root         4096 Jan 16 22:22 usr/local/bin
drwxr-xr-x    3 root     root         4096 Jan 16 19:17 usr/local/lib
drwxr-xr-x    2 root     root         4096 Dec 28 00:44 usr/local/sbin
drwxr-xr-x    2 root     root         8192 Dec 27 02:10 usr/sbin
```

We will use while loops further in the Test and Case sections.

Test

Test is used by virtually every shell script written. It may not seem that way, because test is not often called directly. test is more frequently called as [. [is a symbolic link to test, just to make shell programs more readable.

If is also normally a shell builtin (which means that the shell itself will interpret [as meaning test, even if your Unix environment is set up differently):

```
$ type [
[ is a shell builtin

$ which [
/usr/bin/[

$ ls -l /usr/bin/[
lrwxrwxrwx    1 root     root            4 Mar 27  2000 /usr/bin/[ -> test
```

Test is a simple but powerful comparison utility. For full details, run man test on your system, but here are some usages and typical examples.

Test is most often invoked indirectly via the if and while statements. It is also the reason you will come into difficulties if you create a program called test and try to run it, as this shell builtin will be called instead of your program!

The syntax for if...then...else... is:

```
if [ ... ]
then
  # if-code
else
  # else-code
fi
```

Note that fi is if backwards! This is used again later with case and esac.

Also, be aware of the syntax - the "if [...]" and the "then" commands must be on different lines. Alternatively, the semicolon ";" can seperate them:

```
if [ ... ]; then
  # do something
fi
```

Try the following code snippet, before running it set the variable X to various values (try -1, 0, 1, hello, bye, etc).

You can do this as follows :

```
$ X=5

$ export X

$ ./test.sh
  ... output of test.sh ...
$ X=hello

$ ./test.sh
  ... output of test.sh ...

$ X=test.sh

$ ./test.sh
  ... output of test.sh ...
```

Then try it again, with $X as the name of an existing file, such as /etc/hosts.

test.sh

```
#!/bin/sh
if [ "$X" -lt "0" ]
then
  echo "X is less than zero"
fi
if [ "$X" -gt "0" ]; then
  echo "X is more than zero"
fi
[ "$X" -le "0" ] && echo "X is less than or equal to  zero"
[ "$X" -ge "0" ] && echo "X is more than or equal to zero"
[ "$X" = "0" ] && echo "X is the string or number \"0\""
[ "$X" = "hello" ] && echo "X matches the string \"hello\""
[ "$X" != "hello" ] && echo "X is not the string \"hello\""
[ -n "$X" ] && echo "X is of nonzero length"
[ -f "$X" ] && echo "X is the path of a real file" || echo "No such file: $X"
[ -x "$X" ] && echo "X is the path of an executable file"
[ "$X" -nt "/etc/passwd" ] && echo "X is a file which is newer than /etc/passwd"
```

This syntax is possible because there is a file (or shell-builtin) called `[` which is linked to `test`. Be careful using this construct, though, as overuse can lead to very hard-to-read code. The if...then...else... structure is much more readable. Use of the [...] construct is recommended for while loops and trivial sanity checks with which you do not want to overly distract the reader.

Note that when you set X to a non-numeric value, the first few comparisons result in the message:

```
test.sh: [: integer expression expected before -lt
test.sh: [: integer expression expected before -gt
test.sh: [: integer expression expected before -le
test.sh: [: integer expression expected before -ge
```

This is because the -lt, -gt, -le, -ge, comparisons are only designed for integers, and do not work on strings. The string comparisons, such as != will happily treat "5" as a string, but there is no sensible way of treating "Hello" as an integer, so the integer comparisons complain.

If you want your shell script to behave more gracefully, you will have to check the contents of the variable before you test it - maybe something like this:

```
echo $X | grep -v [0-9] > /dev/null 2>&1
if [ "$?" -eq "0" ]; then
  # If the grep found something other than 0-9, then it's not an integer.
  echo "Sorry, wanted a number"
else
  # The grep found only 0-9, so it's an integer. We can safely do a test on it.
  if [ "$X" -eq "7" ]; then
    echo "You entered the magic number!"
  fi
fi
```

In this way you can echo a more meaningful message to the user, and exit gracefully.

The $? variable is explained in Variables – Part II, and grep is a complicated beast, but grep -v [0-9] finds lines of text containing anything other than the digits 0-9.

The >/dev/null 2>&1 directs any output or errors to the special "null" device, instead of going to the user's screen.

We can use test in while loops as follows:

test2.sh

```
#!/bin/sh
X=0
while [ -n "$X" ]
do
  echo "Enter some text (RETURN to quit)"
  read X
  echo "You said: $X"
done
```

Alexander Weber has pointed out that running this script will end untidily:

```
$ ./test2.sh
Enter some text (RETURN to quit)
fred
You said: fred
Enter some text (RETURN to quit)
wilma
You said: wilma
Enter some text (RETURN to quit)

You said:
$
```

This can be tidied up with another test within the loop:

test2.sh   modified

```
#!/bin/sh
X=0
while [ -n "$X" ]
do
  echo "Enter some text (RETURN to quit)"
  read X
  if [ -n "$X" ]; then
    echo "You said: $X"
  fi
done
```

Note also that I've used two different syntaxes for if statements on this page. These are:

```
if [ "$X" -lt "0" ]
then
  echo "X is less than zero"
fi
```

and

```
if [ ! -n "$X" ]; then
  echo "You said: $X"
fi
```

You must have a break between the if statement and the then construct. This can be a semicolon or a newline, it doesn't matter which, but there must be one or the other between the if and the then.

It would be nice to just say:

```
if [ ! -n "$X" ]
  echo "You said: $X"
```

but the then and fi are absolutely required.

Case

The case statement saves going through a whole set of if .. then .. else statements. Its syntax is really quite simple:

```
case  $VARIABLE in
  option1)
        echo “\$VARIABLE has the value option1”
        ;;
  option2)
        echo “\$VARIABLE has the value option2
        ;;
  *)
        echo “All other undefined options”
        ;;
esac
```

talk.sh

```
#!/bin/sh

echo "Please talk to me ..."
while :
do
  read INPUT_STRING
  case $INPUT_STRING in
        hello)
                echo "Hello yourself!"
                ;;
        bye)
                echo "See you again!"
                break
                ;;
        *)
                echo "Sorry, I don't understand"
                ;;
  esac
done
echo
echo "That's all folks!"
```

Okay, so it's not the best conversationalist in the world; it's only an example!

Try running it and check how it works...

```
$ ./talk.sh
Please talk to me ...
hello
Hello yourself!
what do you think of politics?
Sorry, I don't understand
bye
See you again!

That's all folks!
$
```

The syntax is quite simple:

The `case` line itself is always of the same format, and it means that we are testing the value of the variable INPUT_STRING.

The options we understand are then listed and followed by a right bracket, as `hello)` and `bye)`. This means that if INPUT_STRING matches `hello` then that section of code is executed, up to the double semicolon.

If INPUT_STRING matches `bye` then the goodbye message is printed and the loop exits. Note that if we wanted to exit the script completely then we would use the command `exit` instead of `break`.

The third option here, the `*)`, is the default catch-all condition; it is not required, but is often useful for debugging purposes even if we think we know what values the test variable will have.

The whole case statement is ended with `esac` (case backwards!) then we end the while loop with a `done`.

That's about as complicated as case conditions get, but they can be a very useful and powerful tool. They are often used to parse the parameters passed to a shell script, amongst other uses.

Variables - Part II

There are a set of variables which are already predefined for you, and most of these cannot have values assigned to them.

These can contain useful information, which can be used by the script to know about the environment in which it is running.

The first set of variables we will look at are `$0 .. $9`, `$@`, `$*` and `$#`.

The variable `$0` is the *basename* of the program as it was called.

`$1 .. $9` are the first 9 additional parameters the script was called with.

The variable `$@` is all parameters `$1 .. whatever`.

The variable `$*`, is similar, but does not preserve any whitespace, and quoting, so "File with spaces" becomes "File" "with" "spaces". This is similar to the `echo` stuff we looked at in A first Script.
As a general rule, use `$@` and avoid `$*`.

`$#` is the number of parameters the script was called with.

Let's take an example script:

var3.sh

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

Let's look at running this code and see the output:

```
$ /home/steve/var3.sh
I was called with 0 parameters
My name is /home/steve/var3.sh
My first parameter is
My second parameter is
All parameters are

$
$ ./var3.sh hello world earth
I was called with 3 parameters
My name is ./var3.sh
My first parameter is hello
My second parameter is world
All parameters are hello world earth
```

Note that the value of $0 changes depending on how the script was called. The external utility basename can help tidy this up:

```
echo "My name is `basename $0`"
```

$# and $1 .. $2 are set automatically by the shell.

We can take more than 9 parameters by using the shift command; look at the script below:

var4.sh

```
#!/bin/sh
while [ "$#" -gt "0" ]
do
  echo "\$1 is $1"
  shift
done
```

This script keeps on using shift until $# is down to zero, at which point the list is empty.

Another special variable is $?. This contains the exit value of the last run command.

So the code:

errorcode.sh

```
#!/bin/sh
/usr/local/bin/my-command
if [ "$?" -ne "0" ]; then
  echo "Sorry, we had a problem there!"
fi
```

will attempt to run /usr/local/bin/my-command which should exit with a value of zero if all went well, or a nonzero value on failure.

We can then handle this by checking the value of $? after calling the command. This helps make scripts robust and more intelligent.

Well-behaved applications should return zero on success. Hence the quote:

> One of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate successful termination of their C Programs.

The other two main variables set for you by the environment are `$$` and `$!`. These are both process numbers.

`$$` variable is the PID (Process IDentifier) of the currently running shell.
This can be useful for creating temporary files, such as /tmp/my-script.$$ which is useful if many instances of the script could be run at the same time, and they all need their own temporary files.

The `$!` variable is the PID of the last run background process. This is useful to keep track of the process as it gets on with its job.

Another interesting variable is IFS. This is the *Internal Field Separator*. The default value is SPACE TAB NEWLINE, but if you are changing it, it's easier to take a copy, as shown:

var5.sh

```
#!/bin/sh
old_IFS="$IFS"
IFS=:
echo "Please input three data seperated by colons ..."
read x y z
IFS=$old_IFS
echo "x is $x y is $y z is $z"
```

This script runs like this:

```
$ ./var5.sh
Please input some data seperated by colons ...
hello:how are you:today
x is hello y is how are you z is today
```

External Programs

External programs are often used within shell scripts; there are a few builtin commands (`echo`, `which`, and `test` are commonly builtin), but many useful commands are actually Unix utilities, such as `tr`, `grep`, `expr` and `cut`.

The backtick (`)is also often associated with external commands. Because of this, we will discuss the backtick first.

The backtick is used to indicate that the enclosed text is to be executed as a command. This is quite simple to understand. First, use an interactive shell to read your full name from `/etc/passwd`:

```
$ grep "^${USER}:" /etc/passwd | cut -d: -f5
Steve Parker
```

Now we will grab this output into a variable which we can manipulate more easily:

```
$ MYNAME=`grep "^${USER}:" /etc/passwd | cut -d: -f5`
$ echo $MYNAME
Steve Parker
```

So we see that the backtick simply catches the standard output from any command or set of commands we choose to run. It can also improve performance if you want to run a slow command or set of commands and parse various bits of its output:

external1.sh

```
#!/bin/sh
find / -name "*.html" -print | grep "/index.html$"
find / -name "*.html" -print | grep "/contents.html$"
```

This code could take a long time to run, and we are doing it twice!

A better solution is:

external2.sh

```
#!/bin/sh
HTML_FILES=`find / -name "*.html" -print`
echo $HTML_FILES | grep "/index.html$"
echo $HTML_FILES | grep "/contents.html$"
```

This way, we are only running the slow find once, roughly halving the execution time of the script.

We discuss specific examples further in the Hints and Tipps section of this tutorial.

Functions

One often-overlooked feature of Bourne shell script programming is that you can easily write functions for use within your script.

This is generally done in one of two ways;

- With a simple script, the function is simply declared in the same file as it is called.

- However, when writing a suite of scripts, it is often easier to write a "library" of useful functions, and source that file at the start of the other scripts which use the functions. This will be shown later. (Libraries)

The method is the same however it is done; we will primarily be using the first way here. The second (library) method is basically the same, except that the command

```
. ./library.sh
```

goes at the start of the script.

There could be some confusion about whether to call shell functions *procedures* or *functions.*

The definition of a function is traditionally that is returns a single value, and does not output anything.

A procedure, on the other hand, does not return a value, but may produce output.

A shell function may do neither, either or both. It is generally accepted that in shell scripts they are called functions.

A function may return a value in three ways:

- Change the state of a variable or variables

- Use the `exit` command to end the shell script

- Use the `return` command to end the function, and return the supplied value to the calling section of the shell script

This is rather like C, in that `exit` stops the program, and `return` returns control to the caller.

The difference is that a shell function cannot change its parameters, though it can change global parameters.

A simple script using a function would look like this:

function.sh

```
#!/bin/sh
# A simple script with a function...

add_a_user()
{
  USER=$1
  PASSWORD=$2
  shift; shift;
  # Having shifted twice, the rest is now comments ...
  COMMENTS=$@
  echo "Adding user $USER ..."
  echo useradd -c "$COMMENTS" $USER
  echo passwd $USER $PASSWORD
  echo "Added user $USER ($COMMENTS) with password $PASSWORD"
}

###
# Main body of script starts here
###
echo "Start of script..."
add_a_user bob letmein Bob Holness the presenter
add_a_user fred badpassword Fred Durst the singer
add_a_user bilko worsepassword Sgt. Bilko the role model
echo "End of script..."
```

Line 4 identifies itself as a function declaration by ending in (). This is followed by `{`, and everything following to the matching `}` is taken to be the code of that function.

This code is *not executed* until the function is called. Functions are read in, but basically ignored until they are actually called.

Note that for this example the `useradd` and `passwd` commands have been prefixed with `echo` - this is a useful debugging technique to check that the right commands would be executed. It also means that you can run the script without being root or adding dodgy user accounts to your system!

We have been used to the idea that a shell script is executed sequentially. This is not so with functions.

In this case, the function `add_a_user` is read in and checked for syntax, but not executed until it is explicitly called.

Execution starts with the `echo` statement "Start of script...". The next line, `add_a_user bob letmein Bob Holness` is recognised as a function call so the `add_a_user` function is entered and starts executing with certain additions to the environment:

```
$1=bob
$2=letmein
$3=Bob
$4=Holness
$5=the
$6=presenter
```

So within that function, $1 is set to bob, regardless of what $1 may be set to outside of the function.

So if we want to refer to the "original" $1 *inside* the function, we have to assign a name to it - such as: A=$1 before we call the function. Then, within the function, we can refer to $A.

We use the shift command again to get the $3 and onwards parameters into $@.

The function then adds the user and sets their password. It echoes a comment to that effect, and returns control to the next line of the main code.

Scope of Variables

Programmers used to other languages may be surprised at the scope rules for shell functions.

Basically, there is no scoping, other than the parameters ($1, $2, $@, etc).

Taking the following simple code segment:

scope.sh

```
#!/bin/sh

myfunc()
{
  echo "I was called as : $@"
  x=2
}

### Main script starts here

echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

The script, when called as scope.sh a b c, gives the following output:

```
Script was called with a b c
x is 1
I was called as : 1 2 3
x is 2
```

The $@ parameters are changed within the function to reflect how the function was called.
The variable x, however, is effectively a global variable - myfunc changed it, and that change is still effective when control returns to the main script.

Functions cannot change the values they have been called with, either - this must be done by changing the variables themselves, not the parameters as passed to the script.

An example shows this more clearly:

scope2.sh

```
#!/bin/sh

myfunc()
{
  echo "\$1 is $1"
  echo "\$2 is $2"
  # cannot change $1 - we'd have to say:
  # 1="Goodbye Cruel"
  # which is not a valid syntax. However, we can
  # change $a:
  a="Goodbye Cruel"
}

### Main script starts here

a=Hello
b=World
myfunc $a $b
echo "a is $a"
echo "b is $b"
```

This rather cynical function changes $a, so the message "Hello World" becomes "Goodbye Cruel World".

Recursion

Functions can be recursive - here's a simple example of a factorial function:

factorial.sh

```
#!/bin/sh

factorial()
{
  if [ "$1" -gt "1" ]; then
    i=`expr $1 - 1`
    j=`factorial $i`
    k=`expr $1 \* $j`
    echo $k
  else
    echo 1
  fi
}


while :
do
  echo "Enter a number:"
  read x
  factorial $x
done
```

As promised, we will now briefly discuss using libraries between shell scripts. These can also be used to define common variables, as we shall see.

Libraries

- common.lib

```
# common.lib
# Note no #!/bin/sh as this should not spawn an extra shell.
# It's not the end of the world to have one, but clearer not to.
#
STD_MSG="About to rename some files..."

rename()
{
  # expects to be called as: rename .txt .bak
  FROM=$1
  TO=$2

  for i in *$FROM
  do
    j=`basename $i $FROM`
    mv $i ${j}$TO
  done
}
```

- function2.sh

```
#!/bin/sh
# function2.sh
. ./common.lib
echo $STD_MSG
rename txt bak
```

- function3.sh

```
#!/bin/sh
# function3.sh
. ./common.lib
echo $STD_MSG
rename html html-bak
```

Here we see two user shell scripts, function2.sh and function3.sh, each *source*ing the common library file common.lib, and using variables and functions declared in that file.
This is nothing too earth-shattering, just an example of how code reuse can be done in shell programming.

Exit Codes

For details about exit codes, see the Exit Codes part of the Hints and Tips section of the tutorial. For now, though we shall briefly look at the `return` call.

```
#!/bin/sh

adduser()
{
  USER=$1
  PASSWD=$1
  shift ; shift
  COMMENTS=$@
  useradd -c "${COMMENTS}" $USER
  if [ "$?" -ne "0" ]; then
    echo "Useradd failed"
    return 1
  fi
  passwd $USER $PASSWD
  if [ "$?" -ne "0" ]; then
    echo "Setting password failed"
    return 2
  fi
  echo "Added user $USER ($COMMENTS) with password $PASSWORD"
}

## Main script starts here

adduser bob letmein Bob Holness, the famous BlockBusters presenter!
if [ "$?" -eq "1" ]; then
  echo "Something went wrong with useradd"
else if [ "$?" -eq "2" ]; then
  echo "Something went wrong with passwd"
else
  echo "Bob Holness added to the system."
fi
```

This script checks the two external calls it makes (`useradd` and `passwd`), and lets the user know if they fail. The function then defines a return code of 1 to indicate any problem with `useradd`, and 2 to indicate any problem with `passwd`. That way, the calling script knows where the problem lay.

Anhang

Exit Codes

Exit codes are a number between 0 and 255, which is returned by any Unix command when it returns control to its parent process.
Other numbers can be used, but these are treated modulo 255, so `exit -10` is equivalent to `exit 246`, and `exit 257` is equivalent to `exit 1`.

These can be used within a shell script to change the flow of execution depending on the success or failure of commands executed. This was briefly introduced in Variables - Part II. Here we shall look in more detail in the available interpretations of exit codes.

Success is traditionally represented with `exit 0`; *failure* is normally indicated with a non-zero exit-code. This value can indicate different reasons for failure.
For example, GNU `grep` returns `0` on success, `1` if no matches were found, and `2` for other errors (syntax errors, nonexistant input files, etc).

We shall look at three different methods for checking error status, and discuss the pros and cons of each approach.

Firstly, the simple approach:

```
#!/bin/sh
# First attempt at checking return codes
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
if [ "$?" -ne "0" ]; then
  echo "Sorry, cannot find user ${1} in /etc/passwd"
  exit 1
fi
NAME=`grep "^${1}:" /etc/passwd|cut -d":" -f5`
HOMEDIR=`grep "^${1}:" /etc/passwd|cut -d":" -f6`

echo "USERNAME: $USERNAME"
echo "NAME: $NAME"
echo "HOMEDIR: $HOMEDIR"
```

This script works fine if you supply a valid username in `/etc/passwd`. However, if you enter an invalid code, it does not do what you might at first expect - it keeps running, and just shows:

```
USERNAME:
NAME:
HOMEDIR:
```

Why is this? As mentioned, the `$?` variable is set to the return code of the *last executed command*. In this case, that is `cut`. `cut` had no problems which it feels like reporting - as far as I can tell from testing it, and reading the documentation, `cut` returns zero whatever happens! It was fed an empty string, and did its job - returned the first field of its input, which just happened to be the empty string.

So what do we do? If we have an error here, `grep` will report it, not `cut`. Therefore, we have to test `grep`'s return code, not `cut`'s.

```
#!/bin/sh
# Second attempt at checking return codes
grep "^${1}:" /etc/passwd > /dev/null 2>&1
if [ "$?" -ne "0" ]; then
  echo "Sorry, cannot find user ${1} in /etc/passwd"
  exit 1
fi
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
NAME=`grep "^${1}:" /etc/passwd|cut -d":" -f5`
HOMEDIR=`grep "^${1}:" /etc/passwd|cut -d":" -f6`

echo "USERNAME: $USERNAME"
echo "NAME: $NAME"
echo "HOMEDIR: $HOMEDIR"
```

This fixes the problem for us, though at the expense of slightly longer code.
That is the basic way which textbooks might show you, but it is far from being all there is to know about error-checking in shell scripts. This method may not be the most suitable to your particular command-sequence, or may be unmaintainable. Below, we shall investigate two alternative approaches.

As a second approach, we can tidy this somewhat by putting the test into a seperate function, instead of littering the code with lots of 4-line tests:

```
#!/bin/sh
# A Tidier approach

check_errs()
{
  # Function. Parameter 1 is the return code
  # Para. 2 is text to display on failure.
  if [ "${1}" -ne "0" ]; then
    echo "ERROR # ${1} : ${2}"
    # as a bonus, make our script exit with the right error code.
    exit ${1}
  fi
}

### main script starts here ###

grep "^${1}:" /etc/passwd > /dev/null 2>&1
check_errs $? "User ${1} not found in /etc/passwd"
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
check_errs $? "Cut returned an error"
echo "USERNAME: $USERNAME"
check_errs $? "echo returned an error - very strange!"
```

This allows us to test for errors 3 times, with customised error messages, without having to write 3 individual tests. By writing the test routine once. we can call it as many times as we wish, creating a more intelligent script, at very little expense to the programmer. Perl programmers will recognise this as being similar to the `die` command in Perl.

As a third approach, we shall look at a simpler and cruder method. I tend to use this for building Linux kernels - simple automations which, if they go well, should just get on with it, but when things go wrong, tend to require the operator to do something intelligent (ie, that which a script cannot do!):

```
#!/bin/sh
cd /usr/src/linux && \
make dep && make bzImage && make modules && make modules_install && \
cp arch/i386/boot/bzImage /boot/my-new-kernel && cp System.map /boot && \
echo "Your new kernel awaits, m'lord."
```

This script runs through the various tasks involved in building a Linux kernel (which can take quite a while), and uses the && operator to check for success. To do this with `if` would involve:

```
#!/bin/sh
cd /usr/src/linux
if [ "$?" -eq "0" ]; then
  make dep
    if [ "$?" -eq "0" ]; then
      make bzImage
      if [ "$?" -eq "0" ]; then
        make modules
        if [ "$?" -eq "0" ]; then
          make modules_install
          if [ "$?" -eq "0" ]; then
            cp arch/i386/boot/bzImage /boot/my-new-kernel
            if [ "$?" -eq "0" ]; then
              cp System.map /boot/
              if [ "$?" -eq "0" ]; then
                echo "Your new kernel awaits, m'lord."
              fi
            fi
          fi
        fi
      fi
    fi
  fi
fi
```

... which I, personally, find pretty difficult to follow.

The && and `||` operators are the shell's equivalent of AND and OR tests. These can be thrown together in strings, as above, or:

```
#!/bin/sh
cp /foo /bar && ( echo Success ; echo Success part II ) || ( echo Failed ; echo
Failed part II )
```

This code will either echo

```
Success
Success part II
```

or

```
Failed
Failed part II
```

depending on whether or not the `cp` command was succesful. Look carefully at this; the construct is

```
command && command-to-execute-on-success || command-to-execute-on-failure
```

Only one command can be in each part, though the `( )` brackets make a subshell, which is treated as a single command by the top-level shell.

This method is handy for simple success / fail scenarios, but if you want to check on the status of the `echo` commands themselves, it is easy to quickly become confused about which && and `||` applies to which command. It is also very difficult to maintain. Therefore this construct is only recommended for simple sequencing of commands.