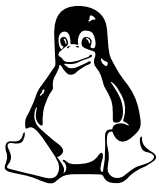

Operating Systems and Systems Integration

# Shell Programming—an Introduction

## Contents

# 1 Aim

After successfully working through this exercise, You will:

- write simple shell scripts using `for`, `if`, `while` statements;
- understand basic regular expressions, and be able to create your own regular expressions;
- understand how to execute and debug these scripts;
- understand some simple shell scripts written by others, and
- be ready to begin to perform automated editing of configuration files using `sed`
- be ready to begin customizing an installation using an automated installation method called `kickstart`, which will be our next laboratory topic.

# 2 Background

A working knowledge of shell scripting is essential to everyone wishing to become reasonably adept at system administration, even if they do not anticipate ever having to actually write a script. Consider that as a Linux machine boots up, it executes the shell scripts in `/etc/rc.d` to restore the system configuration and set up services. A detailed understanding of these startup scripts is important for analyzing the behaviour of a system, and possibly modifying it.

Writing shell scripts is not hard to learn, since the scripts can be built in bite-sized sections and there is only a fairly small set of shell-specific operators and options to learn. The syntax is simple and straightforward, similar to that of invoking and chaining together utilities at the command line, and there are only a few "rules" to learn. Most short scripts work right the first time, and debugging even the longer ones is straightforward.

A shell script is a "quick and dirty" method of prototyping a complex application. Getting even a limited subset of the functionality to work in a shell script, even if slowly, is often a useful first stage in project development. This way, the structure of the application can be tested and played with, and the major pitfalls found before proceeding to the final coding in C, C++, Java, or Perl.

Shell scripting hearkens back to the classical UNIX philosophy of breaking complex projects into simpler subtasks, of chaining together components and utilities.

## 2.1 Where to get more information

There is a free on-line book about shell programming at: `http://www.linuxdoc.org/LDP/abs/html/index.html` and `http://www.linuxdoc.org/LDP/abs/abs-guide.pdf`. The handy reference to shell programming is:

```
$ pinfo bash
```

or

```
$ man bash
```

# 3 The Shebang

A shell script is started by the Linux kernel. The kernel reads the first two bytes of the executable file to determine how to execute it. If it starts with the characters "`#!`" then the kernel will consider this to be executed as a script, run by an interpreter. The kernel then reads the next characters after the "`#!`" to determine what interpreter to use.

For shell scripts, the interpreter is `/bin/sh`, so the first line of all our shell scripts is:

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

If you make any typing mistake in the name of the interpreter, you will get an error message such as "bad interpreter: No such file or directory."

# 4 Making the script executable

To easily execute a script, it should:

- be on the `PATH`
- have execute permission.

How to do each of these?

- Red Hat Linux by default, includes the directory ∼`/bin` on the `PATH`, so create this directory, and put your scripts there.
- If your script is called `script`, then this command will make it executable:

  ```
  $ chmod +x script
  ```

# 5 True and False

Shell programming uses external programs very much. When program execution is successful, programs have an *exit status* of 0, and a non-zero error code when not successful. As a result, shell programming uses the value 0 as true, and non-zero as false.

# 6 Shell Variables

When using the value of a variable, the variable starts with a dollar sign '`$`' When assigning a value to a variable, the variable has no dollar sign. An assignment has no spaces either side of the '`=`':

```
a=375
hello=$a
PATH="$PATH:/sbin:/usr/sbin"
```

## 6.1 Baby Can't Change Parent

Nonsense, any parent will tell me. Okay, I'm talking about processes, not humans. When a parent process `fork()`s and has a child process, the child process inherits all the environment variables of the parent. But the child process cannot change any environment variable of the parent.

If you write a shell script that sets some environment variables and then exits, you will find that all these new values have disappeared. This applies to subshells too, so values set in a subshell are "local". To execute some commands in a subshell, put parentheses around them. See the example in section 7. Here is an example of what I am talking about:

```
$ echo $HOME
/home/nicku
$ pwd
$ cat baby
#! /bin/sh
cd /usr
HOME="Tsing Yi"
echo $HOME
pwd
$ ./baby
Tsing Yi
/usr
$ echo $HOME
/home/nicku
$ pwd
/home/nicku/teaching/ict/ossi/lab/shell
```

# 7 Special Variables

Parameters may be passed to a shell script. '`$0`' is the name of the shell script itself. The first parameter is called '`$1`', the second is '`$2`' and so on. The number of parameters is '`$#`'. A list of all the parameters is in the variables '`$*`' and '`$@`'. The only difference between '`$*`' and '`$@`' is when they are enclosed in double quotes—see section 9.1 on page 6 on quoting.

`IFS` is the "*internal-field separator*". The shell automatically splits strings into fields divided by the `IFS`. Here is a simple example:

```
$ (IFS=:; echo $PATH)
/usr/kerberos/bin /usr/local/bin /usr/bin /bin /usr/bin/X11 /usr/games
/usr/bin /usr/X11R6/bin /opt/OpenNMS/bin /usr/java/jdk1.3.1_01/bin
/home/nicku/bin /sbin /usr/sbin /usr/local/sbin
```

I changed `IFS` in a subshell so that the value of `IFS` in the current shell would not be changed. Sort of like a local variable.

# 8 Special Characters

Comments start with a '`#`'. Statements are separated either by newlines, or by semicolons '`;`'.

The dot command is useful for executing a login script:

```
. ~/.bash_profile
```

It is useful here, because it does not execute the commands in a separate subshell. Hence, all changes to variables remain.

The '$' symbol indicates that a variable name comes next, and gives the value of that variable.

The backslash "\" has many meanings, mostly similar to it's behaviour in the C programming language. At the end of a line, a backslash allows a long line to be split into shorter pieces.

There are many other characters that are special to the shell. See chapter 4 of `http://www.linuxdoc.org/LDP/abs/html/index.html`.

# 9 Quoting

There are four main ways of quoting: forward single quotes, double quotes, the backslash, and backward single quotes. Quoting causes the quoted material to have a different meaning from normal. In particular, the special treatment the shell gives to special characters is suppressed to some degree.

Enclosing in double quotes `"..."` suppresses all special behaviour, except for variable interpretation (`$`), the forward quote, and the backslash.

Enclosing in single forward quotes `'...'` suppresses the special behaviour of all special characters.

Putting a backslash in front of a character preserves the literal value of the character, except for newline.

Single back quotes `` `...` `` mean: "execute the external program called within these quotes and put the output back here." This is called *command substitution* in the bash manual. Command expansion is really quite different from the other three quoting methods. Here is an example using the `hostname` command, which prints the hostname on standard output:

```
$ hostname
nickpc.tyict.vtc.edu.hk
$ h=hostname
$ echo $h
hostname
$ h=`hostname`
$ echo $h
nickpc.tyict.vtc.edu.hk
```

## 9.1 When to use quoting

Many programs, such as `grep` or `find` need some special characters that they themselves will interpret. We need to be able to send these characters unchanged to the program. In this case, quote them. Examples:

```
$ find . -name "*.rpm"
```

If we do not quote the asterisk, the shell will expand `*.rpm` to match only the `rpm` files in the current directory, but we want find to locate all the `.rpm` files in the directories *below* the current directory also.

If you want a variable value that contains spaces to not be automatically split my the shell, then quote it. Here, `testquote` is a short shell script that prints information about its parameters:

```
$ test="one two"
$ testquote $test
You have 2 parameters.  They are:
parameter 1: one
parameter 2: two
$ testquote "$test"
You have 1 parameters.  They are:
parameter 1: one two
```

Note that `"$*"` is one value (not split up), while `"$@"` is split into the original parameters. So if '`$#`' had the value 4, then there are four separately quoted values in `"$@"`. See the beginning of section 7 on page 4.

Here is a little example showing the difference between `"$@"` and `"$*"`:

```
$ cat test_at_star
#! /bin/sh
testquote "$@"
testquote "$*"
$ test_at_star one two three
You have 3 parameters.  They are:
parameter 1: one
parameter 2: two
parameter 3: three
You have 1 parameters.  They are:
parameter 1: one two three
```

Notice how `"$*"` just turned into one long parameter that contains spaces.

## 9.2 Printing Output

We use `echo` to print things. By default, it puts a new line at the end. To avoid printing a newline, use `echo -n`:

```
$ cat echo-n
#! /bin/sh
echo "Hello "
echo World
echo -n "Hello "
echo World
$ ./echo-n
Hello
World
Hello World
```

## 9.3 Reading Input

There are many ways of reading input, but one simple way is to use `read`;

```
$ read answer
yes
$ echo $answer
yes
```

# 10 The Basic Statements

The shell is a complete programming language, and supports `for` loops, `while` loops, `if` statements, `case` statements (like `switch` in C), as well as function calls. We look at only a small subset of these.

## 10.1 The `if` statement

The syntax of the `if` statement is:

```
if test-commands
then
    statements
fi
```

We can add an *else*:

```
if test-commands
then
    statements-if-true
else
    statements-if-false
fi
```

and we can have other `if` conditions nested inside, but they are introduced with a new keyword: `elif`:

```
if test-commands
then
    statements-if-test-commands-1-true
elif test-commands-2
then
    statements-if-test-commands-2-true
else
    statements-if-all-test-commands-false
fi
```

The `test-commands` is either:

- a program being executed, or
- a test made using the program `test`; see `man test` for all the tests you can make using `test`. Also see section 10.5 on page 9.

A simple example:

```
if grep nick /etc/passwd > /dev/null 2>&1
then
    echo Nick has a local account here
else
    echo Nick has no local account here
fi
```

We redirect all output from grep to avoid the side effect of printing the line grep found.

If you want to put the `then` on the same line as the `if`, you need to put a semicolon before the `then`. Here is another example that adds the user `nicku` to the sudoers file if that user is not there already:

```
if ! grep nicku /etc/sudoers > /dev/null 2>&1; then
    echo "nicku   ALL=(ALL) ALL" >> /etc/sudoers
fi
```

## 10.2 The `while` statement

The format of the `while` statement is:

```
while test-commands
do
    loop-body-statements
done
```

Again, if you want to put the `do` on the same line as the `while`, then you need an extra semicolon before the `do`. A simple example:

```
i=0
while [ "$i" -lt 10 ]; do
    echo -n "$i "       # -n suppresses newline.
    i=‘expr $i + 1‘     # i=$(($i+1)) also works.
done
```

The square brackets are an example of the `test` program. See section 10.5 on the next page.

### 10.2.1 `expr`

In the last example using a `while` loop, we used the program `expr` to do arithmetic. This is the portable way to do arithmetic in shell programming. Note that since `expr` prints its output on standard output, we use command substitution to assign the program output to the variable `i`. See the manual page for `expr` for more information.

## 10.3 The `for` statement

The format of the `for` statement is:

```
for name in words
do
    loop-body-statements
done
```

Here is a simple example:

```
for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
    echo $planet
done
```

You can leave the `in` *`words`* out; in that case, *`name`* is set to each parameter in turn.

Here is another example:

```
for i in *.txt
do
    echo $i
    grep 'lost treasure' $i
done
```

Note that the shell will expand the wildcard characters into a list of file names that you can process one by one in the loop.

## 10.4 `break` and `continue`

Inside loops you can the `break` and `continue` statements. They work like they do in C.

## 10.5 The `test` program

The `test` program is used to perform comparisons of strings, numbers and files, often used with the `if` and `while` statements. I will not waste space by copying the manual page here: do `man test` to read all about `test`.

You can call the test program two ways: one as the name `test`, the other (more common way) as `[ ... ]`. If we look, there is a program called "`[`":

```
$ which [
/usr/bin/[
$ ls -l /usr/bin/[
lrwxrwxrwx    1 root     root            4 Nov 25 13:36 /usr/bin/[ -> test
```

**Important Note:** there must be white space before and after the "`[`":

```
i=0
while ["$i" -lt 10]; do
    echo -n "$i "
    i=`expr $i + 1`
done
bash: [0: command not found
```

## 10.6 Using the "`&&`" and "`||`" Operators for Flow Control

The shell has many operators; see the manual for a complete list. But here we look at two familiar operators that are surprisingly useful, and yet which may have a use that is unfamiliar to you.

They are used for logical operations, and are shortcut logical operators, just as you are familiar with in the C and Java programming languages. However, in shell programming, they are used for flow control, rather like an `if` statement.

Suppose we have a shell script that we must call with two parameters, and that it should fail if there are fewer or more parameters. We can use the '`&&`' operator after a test and exit. Here is a little shell script that will only accept two parameters and will exit with a help message otherwise:

```
#! /bin/sh

[ $# -ne 2 ] && echo $0 parameter1 parameter2 && exit

echo parameter1 is $1, and parameter2 is $2.
```

So let's run it, first with no parameters, then with two:

```
$ ./two-parameters
./two-parameters parameter1 parameter2
$ ./two-parameters p q
parameter1 is p, and parameter2 is q.
```

The syntax is like this:

*command1* `&&` *command2*

*command2* will execute only if *command1* is successful.

Similarly, the syntax for the '`||`' operator is:

*command1* `||` *command2*

*command2* will execute only if *command1* is *not* successful.

# 11 Regular Expressions

Much of what a system administrator does is editing configuration files. There are tools to help with this; one such tool is the program `sed`; another is the programming language Perl. The one thing that comes in useful in both cases are `regular expressions`. The `grep` command also uses regular expressions. Regular expressions provide a way of matching patterns in a text file; they can also provide a way of altering the text that matches the pattern. Getting started with regular expressions is our aim today.

A regular expression is a string of characters. Some of these characters have a special meaning; most do not. The characters with a special meaning are called *metacharacters*. Here are some example regular expressions without metacharacters:

```
/nicku/      # simply matches the string "nicku"
/hacker/     # simply matches the string "hacker"
```

## 11.1 Some Funny Characters (metacharacters)

**Asterisk:** `*` matches zero or more of the thing that came just before. Example:

`1133*` matches 11 followed by one or more 3's, so it will match: 113 or 1133 or 11333 or 1133333333333333...

**Dot:** `.` matches any single character, except newline. So `".*"` matches zero or more of any character.

**Caret:** `^` matches beginning of a line, or inside backets means something different (see below).

**Dollar sign:** `$` matches the end of a line. For example, "`^$`" matches blank lines.

**Brackets:** `[...]` matches one character from the set in the brackets. Examples:

`"[xyz]"` matches the characters `x`, `y`, or `z`.

`"[c-n]"` matches any of the characters in the range `c` to `n`.

`"[B-Pk-y]"` matches any of the characters in the ranges `B` to `P` and `k` to `y`.

`"[a-z0-9]"` matches any lowercase letter or any digit.

`"[^b-d]"` matches all characters *except* those in the range `b` to `d`.

Combined sequences of bracketed characters match common word patterns. `"[Yy][Ee][Ss]"` matches yes, Yes, YES, yEs,...

`"[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]"` matches any IVE student number.

**Backslash:** `\` quotes a metacharacter to take away its special meaning. You can match a literal `"$"` with `"\$"`, or a backslash with `"\\"`.

**Ampersand:** `&` means, in a replacement string, the string to be replaced. See below.

## 11.2 Sed

The `sed` program (**s**tream **ed**itor) is a non-interactive editing program. We will look only at a subset of its behaviour today: substitutions.

Let's start with an example:

```
sed '/nicku/s//nickl/' /tmp/sudoers-orig > /tmp/sudoers
```

On each line of the input file `/tmp/sudoers-orig`, `sed` will replace the first instance of `nicku` with `nickl` and send the result to the output.

Let's pull that expression `/nicku/s//nickl/` apart to see how it works:

It begins with an *address*, which is a simple regular expression without any metacharacters:

```
/nicku/
```

This `sed` address matches all line on the input file that contain the string `nicku`. It will apply the substitute operation to them.

The next part is a *substitution expression*:

```
s//nickl/
```

The syntax of a sustitution expression is:

```
s/pattern to replace/replacement/
```

Here, the *pattern to replace* is empty: that means that we use the value from the address pattern, so we are replacing the string `nicku` with the *replacement*, `nickl`.

So if the input file `/tmp/sudoers-orig` contains this line:

```
nicku   ALL=(ALL) ALL
```

then the output file will contain:

```
nickl   ALL=(ALL) ALL
```

instead.

Here is another example:

```
sed '/^\/misc/s//#&/' /tmp/auto.master-orig > /tmp/auto.master
```

What does this do? It takes as input the file `/tmp/auto.master-orig`, then finds a line starting with `/misc`, and puts a comment character before it. The edited output file is `/tmp/auto.master`.

Again, let us examine this expression `'/^\/misc/s//#&/'` part-by-part:

It begins with an *address*, which (in this case) is a regular expression:

```
/^\/misc/
```

This means: apply the command to lines that start with (the `"^"` metacharacter) `/misc`. We have to use a backslash to quote the forward slash, because otherwise the forward slash would mark the end of the regular expression, rather than a literal forward slash.

The rest is a `substitution expression`:

```
s//#&/
```

Again the *pattern to replace* is empty: that means that we use the value from the address pattern, so we are replacing the string `/misc` with the *replacement*.

The hash symbol `"#"` in the replacement expression is a literal hash, i.e., the comment character that we are inserting.

The special metacharacter `"&"` has the value of the entire *pattern to replace*. So we are replacing a line on the input file like this:

```
/misc   /etc/auto.misc  --timeout 60
```

with this in the output:q

```
#/misc  /etc/auto.misc  --timeout 60
```

## 11.3 Where can I find out more about sed?

The book `http://www.linuxdoc.org/LDP/abs/html/index.html` has an appendix about `sed`. It has a rather limited manual page, but there is an FAQ at `http://www.ptug.org/sed/sedfaq.htm`.

# 12 Finding examples of shell scripts on your computer

Your Linux system has a large number of shell scripts that you can refer to as examples. I counted about 1400. Here is one way of listing their file names:

```
$ file /bin/* /usr/bin/* /usr/sbin/* /sbin/* /etc/rc.d/* /usr/X11R6/bin/* \
| grep -i "shell script" | awk -F: '{print $1}'
```

Let's see how this works. I suggest executing the commands separately to see what they do:

```
$ file /bin/* /usr/bin/*
$ file /bin/* /usr/bin/* | grep -i "shell script"
$ file /bin/* /usr/bin/* | grep -i "shell script" | awk -F: '{print $1}'
```

The `awk` program is actually a complete programming language. It is mainly useful for selecting columns of data from text.

`awk` automatically loops through the input, and divides the input lines into fields. It calls these fields `$1`, `$2`,... `$NF`. `$0` contains the whole line. Here the option `-F:` sets the *field separator* to the colon character. Normally it is any white space. So printing `$1` here prints what comes before the colon, which is the file name.

Suppose you want to look for all shell scripts containing a particular command or statement? Looking for example shell scripts that use the `mktemp` command:

```
$ file /bin/* /usr/bin/* /usr/sbin/* /sbin/* /etc/rc.d/* /usr/X11R6/bin/* \
| grep -i 'shell script'| awk -F: '{print $1}' | xargs grep mktemp
```

Here is a useful little shell script that does this:

```
#! /bin/sh

if [ $# -eq 0 ]
then
    cmd=`basename $0`
    echo $cmd: search all Bourne shell scripts for a command
    echo usage: $cmd [grep-options] command-to-grep-for
    echo the grep-option -l is useful
    exit 1
fi

(
IFS=:
for d in $PATH
do
    file $d/*
done
find /etc/rc.d -type f | xargs file
) \
| grep 'Bourne.* shell script' \
| awk -F: '{print $1}' \
| xargs grep "$@"
```

We run the `for` loop in a sub shell to make the change to `IFS` local. `IFS` is the "internal field separator". The shell will automatically split lines into fields separated by the `IFS`.

## 12.1 Where can I find out more about awk?

There is a whole book about `awk`; you can buy it from O'Reilly for about $300 HK, or you can read it online at `http://www.ssc.com/ssc/eap/`.

# 13 Debugging Shell Scripts

It is best to write shell scripts incrementally: write part, test that it works, and continue until your script does what is required.

You can use *echo* statements to print the values of variables.

You can run the script with the *verbose* option to bash. For a script called `script`, you could run it in verbose mode like this:

```
$ sh -v script
```

You can see each command after it has been expanded by using the `-x` option:

```
$ sh -x script
```

# 14 Common Mistakes

I see many people making the same mistakes. This is due partly to the difference in the shell from other programming languages, and partly due to missing lectures or being late in the laboratory! `:-)`

**Spaces are important!** The shell cares about spaces much more than other programming languages. This is because it does so many different things; if you put

```
i;
```

in a C program, it is just an expression that is evaluated, the result is thrown away, and nothing happens. The shell, on the other hand, will look for an program by the name `i`, and execute it.

The shell breaks things up into separate tokens at white space. Where you put spaces does matter.

**Don't put spaces in assignments:** An assignment is a single thing. If you put spaces in it, the shell will try to execute a program by the name of the variable you are trying to assign to!

```
i =20
bash: i: command not found
```

**`eval` needs spaces:** You need to put spaces between the operands of the external program `eval`:

```
i=0
i=‘eval i+1‘
bash: i+1: command not found
```

**Put spaces around the `[ ... ]`** See the notes in the section 10.5 on page 9.

**Use meaningful variable names:** I saw people get confused about variables such as `$1`, `$2`. Assign them to meaningful names, and you won't get so confused. Use what your other lecturers taught you about good programming practice!

# 15 Questions

Make all these scripts executable programs on your `PATH`.

1. Write a simple shell script that takes any number of arguments on the command line, and prints the arguments with "Hello " in front. For example, if the name of the script is `hello`, then you should be able to run it like this:

   ```
   $ hello Nick Urbanik
   Hello Nick Urbanik
   $ hello Edmund
   Hello Edmund
   ```

2. Write a simple shell script that takes two numbers as parameters and uses a `while` loop to print all the numbers from the first to the second inclusive, each number separated only by a space from the previous number. Example, if the script is called `jot`, then

   ```
   $ jot 2 8
   2 3 4 5 6 7 8
   ```

3. Suppose that the script you wrote for the previous question is called `jot`. Then run it calling `sh` yourself. Notice the difference:

   ```
   sh jot 2 5
   sh -v jot 2 5
   sh -x jot 2 5
   ```

   Do you notice any difference in the output from last two?

4. Write a shell script that, for each `.rpm` file in the current directory, prints the name of the package on a line by itself, then runs `rpm -K` on the package, then prints a blank line, using a `for` loop.

   Mount the server `ictlab.tyict.vtc.edu.hk:/var/ftp/pub` on a convenient directory on your machine, such as `/mnt/ftp`. Test your script on the files in `/mnt/ftp/rh-7.2-updated/RedHat/RPMS`.

   > The option `rpm -K` check**s** that the software package is not corrupted, and is signed by the author (if you have imported the author's public key in your `gpg` setup)

5. Modify the script you wrote for the previous question to print the output of `rpm -K` *only* for *all* the files that fail the test. In particular, if the package's GPG signature fails, then your script should display the output of `rpm -K`. There are at least two packages in this directory which do not have a valid GPG signature; one of them is `redhat-release-7.2-1.noarch.rpm`; what is the other?

   Here is output from `rpm -K` for two packages, one with no GPG signature, the other with:

   ```
   $ rpm -K redhat-release-7.2-1.noarch.rpm bash-2.05-8.i386.rpm
   redhat-release-7.2-1.noarch.rpm: md5 OK
   bash-2.05-8.i386.rpm: md5 gpg OK
   ```

Test it in the same network directory as for the previous question.

**6.** Write a shell script to add a local group called `administrator` if it does not already exist. Do not execute any external program if the `administrator` group already exists.

**7.** Download a copy of the bogus student registration data from `http://ictlab.tyict.vtc.edu.hk/snm/lab/regular-expressions/artificial-student-data.txt`. Use this for the following exercises, together with the `grep` program:

**(a)** Search for all students with the name "CHAN"

**(b)** Search for all students whose student number begins and ends with 9, and with any other digits in between.

**(c)** Search for all student records where the Hong Kong ID has a letter, not a number, in the parentheses.

**(d)** If you have time, you may do the same exercises, but display only the students' names, or student number.

**8.** Write a shell script to take a file name on its command line, and edit it with `sed` so that every instance of "`/usr/local/bin`" is changed to "`/usr/bin`"

**9.** Write a shell script to take a file name on its command line, and edit it using `sed` so that every line that begins with the string `server`:

`server` *`other text`*

is edited so that averything after "`server` " (i.e., the "*`other text`*") is replaced with the string "`clock.tyict.vtc.edu.hk`", so that the line above looks like this:

```
server clock.tyict.vtc.edu.hk
```

Test this on a copy of the file `/etc/ntp.conf` that is on your computer. (Install the package `ntp` if it is not there).