Introduction to Bash Scripts

Scott D. Anderson

Wellesley College
sanderso@wellesley.edu

Draft of September 2, 2002—comments welcome

Abstract

This document is a quick introduction to writing shell scripts using the Bash shell, intended for those who have never written a script before, but who are reasonably familiar with basic Unix commands. (If you've never used Unix before, this document is not for you, but I've written another document introducing Unix.) I also expect that you know a little about programming, so that you're familiar with syntactic and semantic issues. (No handout in the world can cover that!) My goal here is to enable you to read some simple existing scripts and write a few simple ones, but mostly to prepare you to learn more about scripts should you want to.

The "shell" is the command line interface on a Unix™ machine, although the DOS Prompt application under Microsoft Windows is often called a shell, too, since it serves exactly the same purpose. Most people use the shell just to type simple commands like `ls`, `cp`, `rm`, or even slightly more interesting commands like `emacs`, `java` and `netscape`.

Shell programming comes in when we start wanting to do complex sequences of commands. We want to express these sequences in a *language* where the nouns of the language are our computer files and the verbs are commands like the ones above. The shell provides other syntactic elements, like "and," "if," and so forth. In fact, the shell provides an entire programming language, analogous to Java™ or C.

Shell programming, like all programming, begins with a need. This need may be the desire to automate something that is routine, to make things easier for some end user, to avoid a few keystrokes, or to document some procedure. I'll try to motivate each of our examples a little, but not a lot, so try to take on faith that people do use shell programming and find it useful.

1 Command Sequences

Suppose you are in the habit of doing a particular sequence of commands. For example, every time you compile your Java program, you subsequently run it. Therefore, while working on `BuggleBoggle`, you type the following series of commands[1] about 100 times:

```
% javac BuggleBoggle.java
% java  BuggleBoggle
```

Even with filename completion (you *do* use filename completion, right?), your fingers grow weary of this sequence. So, you decide to write a script. You create the following file, which you name something clever like `rejava`. (Don't worry about the first line for now; just always put it in your shell scripts.)

```
#!/bin/bash

javac BuggleBoggle.java
java  BuggleBoggle
```

Now, you put this file in the right place and give it the right permissions,[2] and *voila!,* you've written your first shell script. Okay, it's not earth-shattering, but it's a start.

You use the script just by typing its name, `rejava`, just as if it were a "real" command like `java` or `javac`. Indeed, as far as Unix is concerned, it *is* a real command just like those others. Unix has a philosophy that commands are just programs that it runs, and you have just written a program.

What about compiling your program? You don't: shell programs aren't compiled. Instead, they are *interpreted*, which means that the source code is read and executed as it's read, instead of first being translated (compiled) into another language. Thus, to change a shell script, you just edit the file, save it, and run it. No compilation.

What about the place and permissions that I mentioned? Here at Wellesley, you can put the shell script anywhere. You can run the script by being in the same directory as the script and

[1] I've used the percent sign to stand for your command prompt. Your actual command prompt is probably different, telling you what machine you're on and what directory you're in.

[2] We'll get to these in a minute.

giving its name on the command line. However, you have to tell the computer that the program is *executable*. You can do this by the following incantation:

```
$ chmod u+x rejava
```

Don't worry too much about what this means; essentially, it just means that the program (`rejava`) is now executable by you. If you do a long `ls`, you can see the result, which is the letter "x" in position 4 of the permissions information.

```
$ ls -l rejava
-rwxrw----    1 anderson faculty        56 Aug 23 18:28 rejava
```

To learn more about permissions, read the man page on `chmod`, which is the command you used to change the permissions.

2 Variables

During your next weekly problem set, you soon realize that your script isn't sufficiently general. You'd have to re-write it every time you have a new program to compile. Wouldn't it be nice to be able to supply the filename on the command line, just like you do to programs like `java`? Of course it would. So, the shell automatically provides several variables that are bound to the things on the command line. The variables are numbered from 0 to 9, so if you have more than 9 items on the command line, you have to do other tricks.

The variable is introduced with a dollar sign. The dollar sign rarely appears in the names of commands or files (you now understand that every command is just a filename with the execute permission set), so it easily distinguishes a variable from the other things in a script. The variable that we want is $1. Here's our improved script, which we call `rejava2`:

```
#!/bin/bash

javac $1.java
java  $1
```

Compare this with `rejava`, and you'll see that it's exactly the same except that we globally replaced `BuggleBoggle` with `$1`. That is, we've taken a constant and replaced it with a variable. When the program runs, the variable, `$1` is bound to a constant and the constant is substituted whereever the variable appears.

Notice that the filename that the script uses for `javac` adds the `.java` onto the end. So, we can guess that this script is intended to be used like this:

```
% rejava2 BuggleBoggle
```

Concatenating the `.java` onto the end is automatic, just by writing them next to each other. The shell language makes this kind of thing convenient, because scripts often need to do this.

Soon after that, you'll notice that if you accidentally add the .java onto the end of the filename when you run the script, the script doesn't work. That's because it idiotically adds `.java` onto a filename that doesn't need it, and then discovers that `BuggleBoggle.java.java` doesn't exist. How can this be solved?

Dealing with this problem is common enough that someone wrote a program to strip the suffix off a filename. The stripping is done just as a piece of text; the actual file and filename are unaffected. The program is called `basename`. Here's an example, in `rejava3`:

```
#!/bin/bash

file=‘basename $1 .java‘

javac $file.java
java  $file
```

Notice that `$file` is a variable that we have defined and used. The backticks (it's *very* important to notice that those are backticks and not apostrophes, since the semantics is entirely different) mean to execute the enclosed text as if it were a command and return the result, which we then assign to a variable. We don't have to declare the variable; it's created the first time we assign it a value. Unlike programming languages like Java, shell variables don't have types: the type is always a string.

In `rejava3`, we want to strip off the `.java` if it's there, so we specify it, along with the first command-line argument, to the `basename` program.

3 Conditionals

Your `rejava3` program is pretty nice, but you soon realize that it plunges on and does the second command (executing the Java program) whether or not there were any errors in the first command (compiling the Java program). What we want now is to prevent the second command if the first fails.

What is failure? How can we tell? In Unix, programs can have *return values*, just like methods in Java. (You will increasingly see an analogy between methods in a Java program and programs in a shell script: Java syntax organizes and controls the method execution, just as the shell language organizes and controls program execution.) The return value of a program is just an integer.

Since there are many ways to fail (file not found, permission denied, syntax error and so forth) and often there is only one way to succeed, a convention has been established that a return value of *zero* means success and any non-zero value is failure. Thus, different non-zero return values can be decoded into the reason for the error.

However, in this case, we don't care why if failed, we just care *whether* it failed. We only want to execute `java` if `javac` returns success. Here's our solution, implemented as `rejava4`:

```
#!/bin/bash

file=‘basename $1 .java‘

if ‘javac $file.java‘
then
    java $file
fi
```

Unlike commands such as `java` and `ls`, which are really programs, the `if`, `then`, and `fi` tokens are syntax that is provided by the shell in order to do shell programming.

Our script is really good, but maybe it would be nice to add some "print" statements, so that we can tell the user what is going on. Here's our solution,[3] `rejava5`:

```
#!/bin/bash

file=`basename $1 .java`

if javac $file.java
then
    echo "Compilation of $file.java succeeded"
    java $file
else
    echo "Sorry, compilation of $file.java failed"
fi
```

Notice that you can have an `else` keyword as well as `if` and the others. Also notice the use of `echo` to print strings. There is an `echo` program on every Unix machine, but the shell also has a built-in command for echo (this is for efficiency, so that it doesn't have to execute an external program for something that is so common and easy).

Notice also that you can use variables inside strings and that the values of the variables are automatically *interpolated* into the string. Like the concatenation operation earlier, interpolation is so common and useful that it is a built-in capability of the shell. It's another reason for the ugly dollar-sign syntax for variables: without the dollar sign marker, interpolation would be much harder to do.

Now that we have the ability to use conditionals, we can add something else to our script, namely a "usage" statement. It is a common feature of many Unix programs that they will tell you how to use them, either by running them with no arguments or by giving a special argument such as `-h` or `--help`.[4] Here's an example of how to do that, as `rejava6`:

```
#!/bin/bash

if [ $# -eq 0 ]; then
    echo "Usage: $0 java-file"
    echo "The file will be compiled and executed"
    exit
fi

file=`basename $1 .java`

if javac $file.java
then
```

[3] The backticks around the `javac` execution aren't necessary.

[4] In the olden days, Unix hackers liked being really terse and would use single-letter arguments for almost everything. In more modern times, programmers (particularly the GNU implementers) are trying to be a bit more novice-friendly and are using double-hyphens and words. There are also lots of programs with more than 26 things to configure! Switches like these commonly start with a hyphen, since it's very rare for a filename to start with a hyphen.

```
    echo "Compilation of $file.java succeeded"
    java $file
else
    echo "Sorry, compilation of $file.java failed"
fi
```

There are a couple of new things here. First, you can see that the `exit` keyword can be used to end a script early. Second, the variable `$0`, like the variable `$1`, refers to a command line argument, in this case the name of the command! Third, you can see that you can put the `then` on the previous line as long as there is a semi-colon to mark the end of the condition. Fourth, look at that condition! What is that??

It turns out that there is a Unix program called `test` (which is why you should never name any executable `test`, because you may get the wrong one and confuse yourself to no end). A synonym for the `test` program is `[`. That's right, there's a program whose name is a left square bracket. Weird. The result, however, is just like our other `if` statement, except that the condition is in square brackets. I'll call the program `test`, even though you'll rarely see it called that in shell scripts, where it's almost always referred to by the square bracket.

The `test` program can do lots of things, but it's commonly used for boolean expressions. Not surprisingly, the `-eq` operator says to compare its two operands for equality (as numbers, not as strings). The variable `$#` is a built-in variable like `$1`, and its value is the number of command line arguments. So, what this says is "If the number of arguments is zero, print out this usage message." Here's an example of it in action:

```
% rejava6
Usage: ./rejava6 java-file
The file will be compiled and executed
```

Notice that `$0` was replaced with the name of the script file.

One last example using `test`, because it's also very common. The `test` program can return values depending on whether a file or directory exists, is readable, is executable, and the like. Suppose we decide to be a bit more user-friendly about missing files in our script, we might end up with `rejava7`:

```
#!/bin/bash

if [ $# -eq 0 ]; then
    echo "Usage: $0 java-file"
    echo "The file will be compiled and executed"
    exit
fi

file=‘basename $1 .java‘

if [ ! -f $file.java ]; then
    echo "Sorry, $file.java doesn’t exist; did you mispell the name?"
    exit
fi

if javac $file.java
```

```
then
    echo "Compilation of $file.java succeeded"
    java $file
else
    echo "Sorry, compilation of $file.java failed"
fi
```

Here the `-f` means that the file exists and is a regular file; the exclamation point means negation, so the `echo` occurs if, according to `test`, the file doesn't exist.

4 Loops

This example has moved far beyond usefulness and into hackery, so we should stop adding to it. Instead, let's start fresh with a new script. Very often we want to do something to a whole bunch of files, so let's see how that might be done. Let's write a script that will recompile all the Java files in this directory. Nothing needs to be specified on the command line. (Note that we could add this feature to our earlier script by just checking `$#` and recompiling every file if `$#` indicates that there are no files on the command line. Of course, we give up the ability to do a usage message, unless we assign that to `-h`. We'll leave this as an exercise for the reader. Here's how our script might be done, as `rejava8`:

```
#!/bin/bash

# compiles all java files

for file in `ls *.java`
do
    echo "Compiling $file"
    javac $file
done
```

In this example, we've used the `for` syntax, with additional keywords `in`, `do` and `done`, to loop over a sequence of filenames (as produced by the execution of `ls`) and do something to each one.

(Notice that we used a pound sign to put a comment in the file, since it doesn't have a usage statement.)

One drawback with this version is that we can't be selective about the files we want to compile. It would be nice to specify all the files on the command line and just have the script work on those. There is a special variant of the `for` syntax that leaves out the set to iterate over, implicitly iterating over the command line arguments. Here it is as `rejava9`:

```
#!/bin/bash

# compiles all java files on command line

for file
do
    echo "Compiling $file"
    javac $file
done
```

Using this version, you could recompile all the filenames starting with the letter "B" (but not any others) by the following:

```
$ ls *.java
BuggleBagel.java  BuggleBoggle.java  DoubleBuggle.java
$ rejava9 -all B*.java
Compiling BuggleBagel.java
Compiling BuggleBoggle.java
```

5 Examples

The following example pings each host in a range of IP addresses and records the hostnames of those that respond, thereby giving us a list of machines that are up. The `for` syntax that is shows is pretty rare; I've never seen it in a shell script, but it's there in the man pages. I intended this example to show how awkward arithmetic is in the shell, but it didn't work out that way. Generally speaking, though, you wouldn't use the shell to do arithmetic much more complex than incrementing an integer.

```
#!/bin/bash

# Ping all IP numbers in 149.130.13.*
# Record the hostnames of those that respond

rm live-hosts
for(( n=0 ; $n < 255 ; n=$n+1 ))
do
    ip=149.130.13.$n
    if true #ping -c 1 -w 1 $ip
    then
        host=‘host $ip‘
echo $host >> live-hosts
    fi
done
```

The next example actually interacts with the user, reading inputs from the keyboard and responding to them. It also demonstrates the `case` statement (analogous to the `switch` statement in Java) with its retro-style closing keyword (`esac` is `case` spelled backwards, just as `fi` is the closing keyword for the `if` statement).

```
#!/bin/bash
#

# script to automate use of latex2e.  Implemented at UMass.
# Rewritten in Bash by me

# -- Scott D. Anderson August 2002
```

```
if [ $# -eq 0 ]; then
    echo "Usage: $0 file"
    echo "runs latex and optionally dvips, xdvi, bibtex, and lpr"
    exit
fi

cd `dirname $1`

FILENAME=`basename $1 .tex`

latex ${FILENAME}

while true
do
   echo "-----------------------------------------"
   echo "Enter 'd' to create PostScript file"
   echo "Enter 'v' to preview on screen"
   echo "Enter 'r' to Re-LaTeX the file"
   echo "Enter 'b' to BibTeX the file"
   echo "Enter 'p' to print the PostScript file"
   echo -n "else enter 'e' to exit "

   read choice
   echo $choice
   case $choice in

    d) dvips -o ${FILENAME}.ps ${FILENAME} ;;

    v) xdvi -s 8 ${FILENAME} ;;

    r) latex ${FILENAME} ;;

    b) bibtex ${FILENAME} ;;

    p) lpr ${FILENAME}.ps ;;

    e) exit ;;

    *) echo "no such option" ;;

    esac
done
```

The next example is used to create drop folders for a CS111 student. I actually wrote it very quickly and I have confidence that it will save me and the other system administrators a lot of time at the beginning of the semester. More importantly, I know that the drop folders are all created in a consistent way, so that I can be sure everything has the correct ownership and permissions and such. Even more than speed, shell scripts can capture and document how things work. This script

is called `create-cs111-drop-folders-1`.

```
#!/bin/bash

# This creates drop folders for cs111, the way they like it.  It only
# makes them for one student, say for a straggler who didn't sign up
# and get processed with everyone else.  Generally, you should use
# create-cs111-drop-folders to create them en masse.

# Scott D. Anderson
# September 2002

# This is the number of problem set subdirectories to create.

numps=10

if [ $# -ne 1 ]
then
    echo "Usage: $0 student-username"
    exit 0
fi

# bail out if there are any errors
set -e

user=$1
course=cs111

if [ ! -d /home/$course ]; then
    echo "Course directory doesn't exist!"
    exit 1
fi

if [ ! -d /home/$course/drop ]; then
    echo "course drop directory doesn't exist!"
    echo "Run create-cs111-drop-folders-all"
    exit 2
fi

top=/students/$user/$course
drop=$top/drop

/bin/mkdir $top
/bin/mkdir $drop
/bin/chmod 0755 $top $drop
/bin/chown $user.$user $top $drop

/bin/ln -s $drop /home/$course/drop/$user
```

```
# Make problem set subdirectories
for(( i=1 ; $i <= $numps ; i=$i+1 ))
do
  dir="$drop/ps$i"
  /bin/mkdir $dir
  /bin/chmod 0755 $dir
  /bin/chown $user.$user $dir
done

exit 0
```

6 At The Prompt

So far, we have always written our shell scripts in files, just like regular programs. However, the shell understands its own syntax at the command line, so if you get good at it, you can type complex commands right into the shell and have them execute immediately. For example, to create a backup copy of each .java file in a directory, you could do:

```
% for file in *.java ; do cp $file $file.bak ; done
```

You can't do this without the `for` syntax, because the following wouldn't work:

```
cp *.java *.java.bak     # won't work
```

7 Return Values

Shell programming is valuable enough that Unix programs almost always return a useful value, so that a shell script can at least determine whether the program "succeeded" or "failed." For example, the `ping` program, which we used above, has the following behavior (quoted from the man page):

> If ping does not receive any reply packets at all it will exit with code 1. If a packet count and deadline are both specified, and fewer than count packets are received by the time the deadline has arrived, it will also exit with code 1. On other error it exits with code 2. Otherwise it exits with code 0. This makes it possible to use the exit code to see if a host is alive or not.

Thus, to be a good citizen, Unix programs should return values. In C, you do that like this:

```
int main( int argc, char* argv[] ) {
    ...
    return 0;
}
```

The `argc` parameter is the number of command-line parameters, which is the `$#` shell variable. The `argv` parameter is an array of strings, just like `String[]` in Java.

What about shell scripts? A shell script returns a value by supplying one with the `exit` keyword. For example:

```
if [ $# -eq 0 ]; then
    echo "Usage: $0 files"
    exit 0
fi
```

We omitted them in this document so that we had less to explain.

8 Security

If you're writing shell scripts for others to use, you will need to worry about security. There are certain security issues that apply to shells scripts as well as ordinary binary files and others that apply only or particularly to shell scripts. Security is a difficult, multi-faceted problem, so please remember that I've just scratched the surface in this sections.

8.1 Suid Scripts

Usually, a program runs with the permissions of the person running it. So, for example, even though the `cat` program (`/bin/cat` is owned by `root`, when you use it, you can only read the contents of files that you have read permission to. This is the normal, expected state of affairs.

There are times, though, when we need to circumvent this. Consider email: you want to send email to Sally. However, you don't have write permission to sally's mailbox (`/var/mail/sally`). Similarly, Sally doesn't have read permissions on a file in your account, so you can't leave the file someplace for her to get. One solution is to give the email message to an intermediary that *does* have permission to write to Sally's mailbox. Often, that intermediary is the omnipotent user called *root*. In short, when you send an email message, you are executing a program (`sendmail`) that is running not with your privileges and permissions, but with root's. Such a program is called *setuid* (or *suid*). Let's look at its permissions:

```
ls -l /usr/sbin/sendmail
-r-sr-xr-x    1 root     root       417828 Mar  3  2001 /usr/sbin/sendmail
```

Notice the *s* where you normally would see an *x*. That means the program runs *setuid*. Setuid programs run with the permissions of the *owner* of the program, in this case, `root`.

Now, let's see some of the security implications of a setuid shell script. Suppose you're root and the users are complaining that novices are sending binary files to the printer, resulting in 300+ pages of garbage output, which is a waste of time, paper and other things. They can't remove the print jobs because they aren't owners of the files.

You decide to make it possible for them to delete others print jobs[5]. You call it `lprm-other` and toss it off—it's the world's easiest shell script:

```
#!/bin/bash

lprm $1
```

Of course, it needs to be *setuid* root to work, since root can delete anyone's files. You can accomplish this:

[5]The Right Thing might be to implement a print command (as a shell script) that detects binary files and rejects them. That would be a nice exercise.

```
# chmod 04555 lprm-other
```

Everything is fine, but then you discover that someone has captured your `/etc/shadow` file. What they did was to use your script as follows:

```
% lprm-other "345 ; chmod 666 /etc/shadow /etc/passwd"
```

So, the script ends up executing:

```
# lprm-other 345 ; chmod 666 /etc/shadow /etc/passwd"
```

and it does so as root. It doesn't matter whether the `lprm` succeeds; the important part is that the `chmod` is running as root and so now the `/etc/shadow` file is world readable and writable.

This illustrates the standard setuid exploit: an attacker somehow employs a setuid program to have a command executed with root privileges. Here, they exploited the fact that the script included something from the user on the command line. This leads to one of the primary rules of computer security:

> Never trust anything from the user.

Checking for this kind of thing is difficult to do as a shell script becomes more complex.[6] For this reason, many Unix kernels, including Linux, have forbidden setuid scripts. A setuid program *must* be a binary file. (So, this example doesn't really work on your Linux machine. Sorry.)

8.2 Tainted Data

Here's a slightly different example that illustrates the same issue: don't trust the user. In this case, the system administrator has created an automated way to create accounts, where the name of the account is written in a file (perhaps by some other method). There is no setuid issue here; the system administrator is running the commands by herself, running as root. The file contains entries like:

```
wwellesl : Wendy Wellesley
jjunior  : Janis Junior
ssenior  : Sally Senior
```

The shell script to create the accounts is something like this:

```
#!/bin/bash

for n in `cut -d: -f1 $1`
do
eval "./adduser $n"
done
```

Again, everything seems fine, but then the machine is hacked into. The sysadmin notices that the accounts file is:

---

[6] Perl, a different scripting language, has "taint" checking that aids this greatly, and is therefore preferred for any security-conscious scripting.

```
wwellesl : Wendy Wellesley
jjunior  : Janis Junior
ssenior  : Sally Senior
joe;/tmp/mychmod : joe k00l
```

Hmm, that account `joe;/tmp/mychmod` is pretty odd. What is `/tmp/mychmod`? If joe hasn't removed the file already, it might look like:

```
#!/bin/bash

chmod 666 /etc/shadow /etc/passwd
```

This simple script makes[7] a copy of the `/bin/bash` (the shell) and makes it setuid, so now anyone running `/tmp/tempfile` becomes root!

This example reinforces the rule, which I will repeat at the risk of being boring:

> Never trust anything from the user.

8.3 A Secure Path

Another exploit that a user can do is to trick the system administrator into running a command that is bound to something different from he or she expected. That sentence is pretty abstract, so let's be specific.

Suppose that `root` has "." in its path—dot is Unix-speak for the current working directory. (This is a bad idea, for reasons we are about to see.) Suppose also that root has a shell script to list the most recently modified files in a directory.[8]

```
#!/bin/bash

ls -lt | head
```

Notice that two programs are being run: `ls` and `head`. Since "." is in the path, if a file with either of those names is in the current directory, it might be run instead of the "real" `ls` and `head` (which are `/bin/ls` and `/usr/bin/head`.

Our nefarious user just creates the following shell script, names it `ls` and puts it in a directory.

```
#!/bin/bash

# To avoid error messages that will give us away, don't even try this if
# we're not root.

if [ $UID -eq 0 ]; then
   chmod 666 /etc/shadow /etc/passwd
fi
```

---

[7] Unfortunately, I wasn't able to get this exact example to work properly, but I believe that something like it will work and that the general point is still true.

[8] This script could also be an alias, but the point is the same.

```
# Finally, use the real ls to generate the output

/bin/ls $*
```

You can see that if the user sends mail to the system administrator and says "there are some strange new files in this directory, can you tell me what they are?" they might get the system administrator to do:

```
# cd /some/dir
# ls-recent
```

and the damage is done.

How to prevent this? You just have to be sure what programs are running via your shell scripts. There are two ways to do that. The first is to specify a value for the PATH variable in your scripts:

```
#!/bin/bash

PATH="/bin:/usr/bin"

ls -lt | head
```

The other is to explicitly give an absolute pathname for every command:

```
#!/bin/bash

/bin/ls -lt | /usr/bin/head
```

9 Conclusion

Shell scripts are cool. They are essentially programs built out of programs, which is only possible thanks to a Unix philosophy of writing simple programs that do single tasks reasonably efficiently and writing their output to STDOUT, so that the programs can become tools and modules that are combined using shell scripts and pipes and such to do interesting, unanticipated things. Contrast this with a Windows/Macintosh philosphy where every program has to have a GUI and can only interact with the user via keyboard and mouse. It would be very hard to create scripts out of such programs.

Scripts can be very powerful, and like any powerful thing, they can be both useful and dangerous. If you're using them as system administrator, it helps to think about the security issues of them.

Acknowledgments

Thanks to Lyn Turbak for inspiring me to write this.

To Do

You can also program in Csh.

fix the security examples

talk about aliases?

talk about paths?

talk about chmod?