Basic Shell Scripting

BZAN 583

Drew Schmidt

Shell as a Programming Language

The shell, together with many userland utilities such as `[` (people often blur the line between these two distinct things), forms a useful programming language. However, it is a very strange programming language, unlike any you have dealt with in the past. For example, its only data type is character, and it has no data structures to speak of. However, it does have functions.

When we refer to a *shell script*, we generally mean a file containing instructions to be interpreted and executed by the shell (plus userland — we won't bring this up again). Some people denote these files with a `.sh` extension, although most don't bother. Anything you would type into the shell interactively is a good candidate for a shell script. For example, the canonical "hello world" is:

```
#!/bin/sh

echo hello world
```

We will come back to the first line in a moment. The third line is a simple command we are alredy familiar with. We can save this file as `hw.sh` or merely `hw`, say, by entering this text in our preferred editor and saving the file at a place of our choosing. Having done so, you would execute the script by entering

```
./hw # or ./hw.sh, depending on how you named it
```

However, in order for the script to properly execute, you must first change the file's permissions to allow it to be executed. Permissions become fairly complicated and largely go beyond the scope of this course. But to make a file executable, you can enter:

```
chmod +x hw
```

We only have to do this once, and having done so, we can now execute the script as desired.

Finally, of course, the syntax `./hw` executes the script named `hw` in our current working directory, i.e., using a relative path. We can execute from an absolute path, or a different relative path depending on the current working directory. For example, say the file is in `~`. Then we could enter any of:

```
cd ~/.. && ./my_user_name/hw

cd / && ~/hw
```

Shebang

Although generally not strictly necessary, we often add a "shebang" or a "hashbang" to the beginning of a shell script.

Sometimes we legitmately need to use the functionality native to a particular shell. For example, the BASH shell has loop syntax available that many other shells will not parse. If you were to use that syntax, even if your native shell is BASH, it is only polite to declare

```
#!/bin/bash
```

at the start of your script. If you aren't sure about your use of features, you can always set the shebang to your default shell (which is probably BASH). Some people look down on this behavior, but it reasonable people recognize it as a practical compromise.

Exiting

Every program you execute has an "exit status". This is a number that indicates if the program's execution was successful or not. By convention, an exit status of 0 means the execution was a success. Non-zero values mean something went wrong. One can inspect the exit status of the last program with the variable `$?`. So for example:

```
ls > /dev/null # don't display output
echo $?

ls -V 2> /dev/null
echo $?
```

```
## 0
## 2
```

If a program has non-zero exit status, you can find the meaning of the given number in its man page. For example, in the manpage for `ls`, we find:

> Exit status: 0 if OK,
>
> 1 if minor problems (e.g., cannot access subdirectory),
>
> 2 if serious trouble (e.g., cannot access command-line argument).

And indeed, `-V` is not a valid flag for our `ls`.

Variables

We have already seen several special variables so far, namely `$HOME` and `$?`. But we can define new variables easily enough.

The general rules for variables are:

- Assign with `=`
- If a space occurs, put the RHS in quotes
- Reference with preceeding `$`
- Store the output of a command with "backticks" `` ` ``

So for example:

```
x="some text"
echo $x

x=`ls ~ | grep D`
echo $x
```

```
## some text
## Desktop Downloads
```

One thing of interest is that all variables in the shell are character. Strictly speaking, there are no numeric types, although arithmetic is possible in a variety of ways. Simple arithmetic can be performed with a judicious usage of `$` and `((`:

```
echo $((1+2))
```

```
## 3
```

For anything more complicated than this, you need to use a different tool. Many "old timers" use `bc`, which is very cumbersome. Today, most people would use R or Python, though.

When referencing the value of a variable, it is often customary to encase the variable name in brackets. For example, `$HOME` could also be written `${HOME}`. This is giving extra clues to the shell to make sure that things are interpreted correctly. It is often not necessary, but sometimes to avoid ambiguity (and bugs), it is, and so if you are unsure, you can reduce the chance for bugs by adding these.

Finally, we note that "undefined" variables are blanks. To prove this, we an print the output of an undefined variable surrounded by other characters:

```
echo X${definitely_not_defined}X
```

```
## XX
```

This is a very, very important point, which we will return to in a future section.

Loops

Loops are possible in the shell, and the syntax is mostly reminiscent of other programming languages that you are familiar with. The basic `for` loop syntax looks something like:

```
for var in $VAR; do
    ...
done
```

where the `...`'s represent our loop body. Here, the `do` is very important and can not be suppressed. However, we could put it on the following line if we so chose:

```
for var in $VAR
do
    ...
done
```

It's a matter of style and personal preference as to how you wish to style these; either is fine.

Take note of the naming scheme here. `VAR` is an existing, defined variable, and `var` is a *new* variable which we are defining. Since `VAR` is already defined, we must reference it with a `$`. Since `var` is not yet (or is about to be overwritten if it is), we *do not* reference it with a `$`. This might seem strange, but the rules are entirely consistent with naming and referencing variables, as we have already seen.

We can also inline commands instead of storing their outputs first as a variable `VAR`. We do this exactly as you might expect:

```
for var in `command`; do
        ...
done
```

For example, the next three loops have identical behavior:

```
for i in 1 2 3 4 5 6 7 8 9 10; do
        echo $i
done

numbers="1 2 3 4 5 6 7 8 9 10"
for i in $numbers; do
    echo $i
done

for i in `seq 1 10`; do
    echo $i
done
```

Finally, `while` loops are also possible, but they are less useful and beyond the scope of this document.

Logic

Conditionals are possible in the shell, but with a very alien and bizarre syntax and set of rules. We will highlight only some of the more important issues here.

At its most basic, the `if/then/else` construct found in virtually all programming languages takes the form:

```
if [ EXPRESSION ]; then
    ...
elif [ EXPRESSION ]
    ...
else
    ...
fi
```

Here, `if` begins a conditional block, and `fi`, cutely, ends it. As with loops and `for/do`, here we have `if/then`. Beyond the first condition, any additional condition is given by an `elif` declaration, and a fallback condition denoted by `else`. Note that in one of the rare instances for the shell, spacing matters very much in `if` declarations. Such a declaration takes the form:

- `if`
- space

- `[`
- space
- logical expression
- space
- `]`

As one might expect, the usual logical operators exist. In the shell, they take the form:

- Negation: `!`
- And: `EXPRESSION1 -a EXPRESSION2`
- Or: `EXPRESSION1 -o EXPRESSION2`

Unfortunately, logical expressions in the shell are often very cumbersome and unwieldy. In many languages, a simple `==` suffices. Not so in the shell! So say we want to compare two strings. We have:

- String has non-zero length: `-n STRING`
  - `-n s` is `true`
  - `-n ""` is `false`
- Equality: `STRING1 = STRING2`
  - `a = a` is `true`
  - `a = b` is `false`
- Inequality: `STRING1 != STRING2`

Which perhaps looks alright. Except that, as hinted at above, an undefined variable is a "blank", and *you can not test equality of a string with a blank directly.*

Example:

```
[ $this_doesnt_exist = foo ]
echo $?
```

```
## bash: line 0: [: =: unary operator expected
## 2
```

The solution to this problem is to either:

- Always test if a variable is length 0 (`-n`).
- Modify your test:

```
if [ X$this_doesnt_exist = Xfoo ];then
```

Sadly, logical constructions in the shell are unwieldy for yet another reason. Using `=` is only valid for strings! If you want to compare numbers, you would use `-eq` for equality, `-gt` for "greater than", etc. And there are additional flags for, for example, the existence of a file or directory. A common construct is to check if a file exists, and if it does do one thing, and if not do another. A quick example of this is:

```
if [ ! -e foo ]; then
    touch foo
else
    rm foo
fi
```

This checks if the file `foo` exists, and if not (via the `!`), create an empty file named `foo`, and otherwise (if it exists) delete it.

For a full list of these flags, see `man [` or `man test`.