CS2043 - Unix Tools & Scripting

Cornell University, Spring 2014[1]

Instructor: Bruno Abrahao

February 24, 2014

[1] Slides evolved from previous versions by Hussam Abu-Libdeh and David Slater

A note on awk

```
for (item in array)
```

The order in which items are returned is arbitrary.

Recap: Passing arguments to scripts

When we pass arguments to a bash script, we can access them in a very simple way:

- `$1, $2, ...  $10, $11` : are the values of the first, second etc arguments
- `$0` : The name of the script
- `$#` : The number of arguments
- `$*` : All the arguments, "$*" expands to "$1 $2 ... $n",
- `$@` : All the arguments, "$@" expands to "$1" "$2" ... "$n"
- You almost always want to use `$@`
- `$?` : Exit code of the last program executed
- `$$` : current process id.

You have the power!

We now have a variety of UNIX utilities at our disposal and it is time to learn about

scripting!

Scripting 101

Definition:

A script is very similar to a program, although it is usually much simpler to write and it is executed from source code (or byte code) via an interpreter. *Shell scripts* are scripts designed to run within a command shell like `bash`.

Scripts are written in a scripting language, like perl, ruby, python, sed or awk. They are then run using an interpreter. In our case, the scripting language and the interpreter are both **bash**.

If conditionals

If statements are structured just as you would expect:

```
if cmd1
then
    cmd2
    cmd3
elif cmd4
then
    cmd5
else
    cmd6
fi
```

- Each conditional statement evaluates as true if the `cmd` executes successfully (returns an exit code of 0)

Putting it on one line

Sometimes we might want to type a multiline command into the shell, we can do this by hitting enter for each line, or by using semicolons to tell the shell to start new lines:

Example:

```
if [ testexpr ] ; then command1 ; command2 ; fi
```

Exercise

Let's write a script to send us our weekly tasks (hw2, problem 2), which doesn't send us a blank e-mail on weekends.

A simple script

textsearch.sh

```
#!  /bin/bash
# This script searches a file for some text then
# tells the user if it is found or not.
# If it is not found, the text is appended

if grep "$1" $2 > /dev/null
then
    echo "$1 found in file $2"
else
    echo "$1 not found in file $2, appending."
    echo $1 >> $2
fi
```

test expressions

We would not get very far if all we could do was test with exit codes. Fortunately bash has a special set of commands of the form `[ testexp ]` that perform the test **testexp**. First to compare two numbers:

- `n1 -eq n2` : tests if $n1 = n2$
- `n1 -ne n2` : tests if $n1 \neq n2$
- `n1 -lt n2` : tests if $n1 < n2$
- `n1 -le n2` : tests if $n1 \leq n2$
- `n1 -gt n2` : tests if $n1 > n2$
- `n1 -ge n2` : tests if $n1 \geq n2$

If either $n1$ or $n2$ is not a number, the test fails.

Test Expressions

We can use test expressions in two ways:

- `test EXPRESSION`
- `[ EXPRESSION ]`

Either of these commands returns an exit status of 0 if the condition is true, or 1 if it is false.

Use `man test` to learn more about testing expressions

Note: Remember you can check the exit status of the last program using the $? variable.

Example

```
#!  /bin/bash
# Searches a file for two strings and prints which
#is more frequent
# Usage:  ./ifeq.sh <file> string1 string2

arg=‘grep $2 $1 | wc -l‘
arg2=‘grep $3 $1 | wc -l‘
if [ $arg -lt $arg2 ]
then
    echo "$3 is more frequent"
elif [ $arg -eq $arg2 ]
then
    echo "Equally frequent"
else
    echo "$2 is more frequent"
fi
```

string comparison

To perform tests on strings use

- `s1 == s2` : s1 and s2 are identical
- `s1 != s2` : s1 and s2 are different
- `s1` : s1 is not the null string

Make sure you leave spaces! `s1==s2` will fail!

Expansion

When using `testexp` variable substitution is performed, but no matching is perform.

If `x` is the null string, what will `[ $x != monster ]` return?

Expansion

When using `testexp` variable substitution is performed, but no matching is perform.

If `x` is the null string, what will `[ $x != monster ]` return?

It will return an error, because $x is expanded to the null string and the test becomes `[ != monster ]` .

To make sure there are no errors, place your variables inside double quotes. Then
`[ $x != monster ]` is expanded to `[ "" != monster ]` which returns true.

path testing

If **path** is a string indicating a path, we can test if it is a valid path, the type of file it represents and the type of permissions associated with it:

- `-e path` : tests if **path** exists
- `-f path` : tests if **path** is a file
- `-d path` : tests if **path** is a directory
- `-r path` : tests if you have permission to read the file
- `-w path` : tests if you have write permission
- `-x path` : tests if you have execute permission

"A long time ago in a galaxy far, far away... we had hw1!"
Let's understand how it works!

More on testing

You can combine tests:

```
if [ testexp1 -a testexp2 ]
then
    cmd
fi
```

- `-a` : and
- `-o` : or
- `!  testexp1` : not

A note about debugging

To debug your code, invoke the script with the -x option. You will then see all the commands successfully executed:

```
$ bash -x ifeq.sh Frankenstein.txt monster the
++ grep monster Frankenstein.txt
++ wc -l
+ arg=33
++ grep the Frankenstein.xt
++ wc -l
+ arg2=3850
+’[’ 33 -lt 3850 ’]’
+ echo ’the is more frequent’
```

Testing arguments

We can now begin to ensure our scripts get the input we want:

```
if [ -f $1 ]
then
    Perform the action you want
else
    echo "This script needs a file as its input
    dummy!"
fi
```

Arithmetic

A little arithmetic can be useful and BASH can perform all the standard operators

Arithmetic

- a++, a– : Post-increment/decrement
- ++a, –a : Pre-increment/decrement
- a+b, a-b : Addition/subtraction
- a*b, a/b : Multiplication/division
- a%b : Modulu
- a**b : Exponential
- a>b, a<b : Greater than, less than
- a==b, a!=b : Equality/inequality
- =, +=, -= : Assignments

Using Arithmetic Expressions

We have already seen one way to do arithmetic:

Example:

```
echo $((2+5))
7
```

We can also use it as part of a larger command:

The ”Let” Built-In

```
VAR1=2
let VAR2=$VAR1+15
let VAR2++
echo $VAR2
18
```

- `let` evaluates all expressions following the equal sign

The Difference

- all characters between the ((and)) are treated as quoted (no shell expansion)
- The let statement requires there be no spaces **anywhere** (so need to quote)
- Both work only with integers, for real numbers use `bc`.

Example:

```
let "i=i + 1"
i=$(($i + 1))
```