CSE 374: Programming Concepts and Tools

Eric Mullen

Spring 2017

Lecture 4: More Shell Scripts

Homework 1

- Already out, due Thursday night at midnight
- Asks you to run some shell commands
- Remember to use your pocket guide
- Read instructions carefully

Today

We understand most of the bash shell and its “programming language”. Final pieces we’ll consider:

- Shell variables: Defining your own, builtin meanings, and exporting
- Conditional statements
- Arithmetic
- For loops

End with:

- Confusing Bits (some bash-specific; some common to shells)
- Why long shell scripts are a bad idea, etc.

Shell Variables

- We already know a shell has state: current working directory, streams, users, aliases, history.
- Its state also includes shell variables that hold strings.
  – *Always strings* even if they are “123” – but you can do math
- We already saw this a little, with PS1 and PATH

Shell Variables

- How to use:
  - to set a variable: `foo='anything'`
  - to make a new variable: just set it
  - to read a variable: `${foo}`
  - to remove a variable: `unset foo`
  - to see current variables: `set`
- For functions and local variables: see the manual
- All variables are global: can escape to anywhere

Why variables?

- Just like in other languages, they're useful
- Some special variables affect shell operation:
  - PS1
  - PATH
  - many others…
- Some variables only make sense when in a script
  - `$#, $0, $1, $2, … $n, $@, $*, $?`

Export

- If I start another process from my shell, will it see the value of my variables?
  - Answer: it depends
- You can determine whether it is with export
  - `export foo`: foo will be visible to new process
  - `export -n foo`: foo will not be visible
- In practice, you'll see `export foo=SOMETHING`

Export

- Suppose I have a script `a.sh`:

```
export x=6

x=4

./a.sh

echo $x

./a.sh
```

`a.sh`

```
echo $x
export x=12
```

If Statements

- Shell has if, just like java
- Just like other shell things, it's weird


```
if [ $# -gt 1 ]
then
    <do stuff>
    <maybe more stuff>
fi
```

Arithmetic

- Shell variables are always strings, so `k=$i+$j` is not integer addition
- However, `((k=$i+$j))` works, and so does `((k=i+j))`
- So does `let k="$i + $j"`
- In above examples, the shell converts the strings to numbers
  - It won't error on malformed numbers, instead just make it 0

For Loops

- Syntax:

```
for v in x1 x2 x3 … xn
do
 body
done
```

- Execute body n times, with `v` set to `xi` on ith iteration
  - afterwards, `v=xn`
- Why so convenient?
  - Don't have to write out `x1 … xn`, can generate
  - Use "`$@`" for list of argument strings

Quoting

- What does `x=*` do?
- if `x` is set to the string `*`, does `$x` mean `*` or all files in current directory?
- How do you get bash to expand things just enough?
- You could use the manual, or you could just try it
  - `x="*"`
  - `echo x`
  - `echo $x`
  - `echo '$x'` (suppresses all substitutions)
  - `echo "$x"` (suppresses some substitutions)

Ways to get it wrong

- Variable name typo: `oops=7` just makes new variable, `ls $oops` gets empty string (just runs `ls`)
- Use same variable twice: just clobbered `HISTFILE=uhoh`
- Spaces in right hand side: use double quotes or will be separated
- Non-number used as number: turns into 0
- `set foo=stuff` silently does nothing (how you assign in csh)
- many more (to find for yourself)

Bash Programming vs. Java

- Bash
  - “shorter”
  - convenient file-access, file-tests, program execution, pipes
  - crazy quoting rules and syntax
  - also interactive
- Java
  - not as many ways to trip up
  - local variables, modularity, typechecking, array bounds checking, …
  - real data structures, libraries, regular syntax
- If it’s more than 200 lines, don’t do it in bash

Strings

- Suppose foo holds the string hello

	Java	Bash
read variable	foo	$foo
string constant foo	“foo"	foo
assign variable	foo = hi	foo=hi
string concat	foo + “oo”	${foo}oo
convert to number	library call	silent and implicit

- Java: variables are easier. Bash: string constants easier
- Biased towards common use

Shell Programming

- Computer scientists automate, and end up inventing bad languages. Not just bash (consider javascript)
- HW3 will be near the limits of what seems reasonable to do with shell scripting
- Many languages attempt to get the best of both worlds: Perl, Ruby, Python, etc…
- In some way it just gets you hooked on short programs
- Picking bash for this class was partly to show you how bad it can be
- Next: Regular expressions, grep, sed, and others

Bottom Line

- Never do something manually when you could use a script

- Never write a script if you need a large, robust piece of software

- Some programming languages try to blur the line between script and large software, you've seen 2 that don't (Java on one end, Bash on the other)