Lecture 10: More Bash Scripting

CS2042 - UNIX Tools

October 22, 2008

Lecture Outline

Basic Operators

While shell scripts are usually used to automate more complex tasks, occasionally a little arithmetic comes in handy. Here is a partial list of operators that you can use:

Syntax:	Meaning:
a++, a−−	Post-increment/decrement (add/subtract 1)
++a, −−a	Pre-increment/decrement
a+b, a-b	Addition/subtraction
a*b, a/b	Multiplication/division
a%b	Modulo (remainder after dividing)
a**b	Exponential
a>b, a<b	Greater than, less than
a==b, a!=b	Equality, inequality
=, +=, -=	Assignments

Using Arithmetic Expressions

There are two good ways to use arithmetic: as its own operation using variables, or in an expansion as part of a larger command.

The "Let" Built-In

let VAR=$1+15

- Evaluates all following expressions

It is generally good form to use the **$[EXPRESSION]** syntax to perform arithmetic expansions. Note that this only calculates the result of EXPRESSION, and does no tests.

Example:

echo $[323*17]

- 5491

Lecture Outline

Defining Arrays

An array is a variable containing multiple values. There are three different ways to create an array:

Declaring an Array

declare -a arrayname

- Explicit declaration, empty until modified

arrayname[index_number]=value

- Puts *value* in the specified position of a new array

arrayname=(value1 value2 ... valueN)

- Creates an array using the given values, indexed sequentially

Accessing Arrays

Once we have created an array, accessing its individual elements is a little different from standard variables. First, we need to add an *index* to indicate which element we want. Second, we have to add curly braces like this:

- **${arrayname[index]}**

Example:

array=('Cornell University' CS2042 'Intro to Unix')
echo ${array[2]}

- `Intro to Unix`

- The special indices '@' and '*' reference all members of an array.

Lecture Outline

Multiple Options

Let's say we want a conditional with 6 different options, so our script can do 6 different things depending on its first argument.

- Simplest way: an if statement, 4 elifs, and an else
- Is there a better way?

The Case Statement

case EXPRESSION in CASE1) command-list;; CASE2) command-list;; ... CASEN) command-list;; esac

- Attempts to match EXPRESSION to a CASE, then executes the corresponding commands
- CASEs are expressions matching a pattern (using Bash wildcards, or not)
- EXPRESSION can be a variable, command output, or a shell expansion

A Case Example

Example:

```
#! /bin/bash
# This script prints the # of days in the month.
MONTH=$(date +%b)
case $MONTH in
  Jan|Mar|May|Jul|Aug|Oct|Dec)
     NUMDAYS=31;;
  Apr|Jun|Sep|Nov)
     NUMDAYS=30;;
  Feb) NUMDAYS=28;;
esac
echo "This month of $MONTH has $NUMDAYS days."
exit
```

The Select Statement

Here is a simple way to get Bash to make a menu for you:

Example:

```
#! /bin/bash
# Simple example of a select statement
PS3='Choose an option: '
select WORD in "Linux" "Bash" "CS2042" "Cornell"
do
echo "The word you chose is $WORD."
# Break, or else we'll get stuck in a loop
break
done
```

Lecture Outline

Our Simplest Loop

What if we want to repeat a task several times?

- Can just type the commands over and over

Okay, well, what if we want to repeat a task infinitely?

- Either way, use loops!

While Loops

while condition; do command-list; done

- Executes command-list until 'condition' no longer returns true
- When 'condition' fails, the script continues with the command following 'done'
- 'condition' can be any expression or command that returns a status

Until Loops

Syntax

until test-command; do command-list; done

- Executes command-list until test-command returns true
- Same as a while loop with an inverted condition

Example:

while true; do sleep 1; done

- Will loop indefinitely

until false; do sleep 1; done

- So will this!

Lecture Outline

Fixed-Length Loops

Let's say we want to backup each .txt file in a directory by copying it to filename.txt.bak.

For Loops

for name in word; do list; done

- Expands *word* into a list of items
- Replaces *name* with each item as it performs *list*

Example:

for FILE in `ls *.txt`; do cp $FILE $FILE.bak; done

- Adds the .bak extension to copies of all our .txt files

A Good Example

Example:

```
#! /bin/bash
# Reverts our .txt files to their .bak copies
LIST=$(ls *.txt.bak)
for FILE in $LIST; do
# Strip the .bak off our filenames
   file2=$(echo $FILE | sed 's/\.bak//')
   mv $FILE $file2
done
exit 0
```

- This script will replace all our backed up .txts with their .bak counterparts.

Lecture Outline

Why We Need Functions

What is a Function?

Shell functions are a way to group commands together for later execution, using a single name for the group.

Functions provide us with some seriously handy properties:

- Abstraction
  - We can focus on individual building blocks rather than the whole structure
- Modularity
  - Wrote a handy, generalized function? Use it in your other scripts!
- Readability
  - Smaller code blocks are easier to wrap your mind around

Defining Our Own Functions

There are two ways to define functions of your own:

Function Syntax

function name { commands; }
name () { commands; }

- Parentheses are required if the 'function' keyword isn't used
- Spaces between curly braces and commands are required!
- End command list with either a semicolon or a newline.

Lecture Outline

Scope

When you define a variable, its use is limited to a certain context, or *scope.* By default, variables are declared *globally*, meaning that they can be accessed and modified from anywhere in the script. *Local* variables are defined only for the context in which it was created.

Example:

```
VAR="global variable"
function func {
    local VAR="local variable"
    echo $VAR; }
# Execute our new function!
func
echo $VAR
```

Using Function Parameters

We know how to access shell script parameters ($1-$n, remember?). What if we need to pass parameters to a function?

- Use the same variables!
- Anything following a function call is automatically created as a local version of $1-$n

Example:

```
function function_A {
   echo $1; }
function_A "A function parameter!"
echo $1
```

- **./example.sh "A script parameter!"**

```
A function parameter!
A script parameter!
```