CSE 391
Lecture 7

Intro to shell scripting

slides created by Marty Stepp, modified by Jessica Miller, Ruth Anderson and Zorah Fung
http://www.cs.washington.edu/391/

Lecture summary

- basic script syntax and running scripts
- shell variables and types
- control statements: the for loop, if/else, while/until
- arrays
- functions

Shell scripts

- **script**: A short program meant to perform a targeted task.
  - a series of commands combined into one executable file

- **shell script**: A script that is executed by a command-line shell.
  - `bash` (like most shells) has syntax for writing script programs
  - if your script becomes > ~100-150 lines, switch to a real language

- To write a `bash` script (in brief):
  - type one or more commands into a file; save it
  - type a special header in the file to identify it as a script (next slide)
  - enable execute permission on the file
  - run it!

Basic script syntax

#!*interpreter*

- written as the first line of an executable script; causes a file to be treated as a script to be run by the given interpreter
  - (we will use `/bin/bash` as our interpreter)

- Example: A script that removes some files and then lists all files:

```
#!/bin/bash
rm output*.txt
ls -l
```

Tip: The `file` command returns the type of the file, e.g.: `file foo.sh`

```
foo.sh: Bourne-Again shell script, ASCII text executable
```

Running a shell script

- by making it executable (most common; recommended):

```
chmod u+x myscript.sh
./myscript.sh
```

  - fork a process and run commands in `myscript.sh` and exit

- by launching a new shell : (will consult your `.bashrc`)

```
bash myscript.sh
```

  - advantage: can run without execute permission (still need read permission)

- by running it within the current shell:

```
source myscript.sh
```

  - advantage: any variables defined by the script remain in this shell (more on variables later)
  - Will consult your aliases

echo

command	description
`echo`	produces its parameter(s) as output (the `println` of shell scripting) **`-n` flag to remove newline** (`print` vs `println`)

- Example: A script that prints your current directory.

```
#!/bin/bash
echo "This is my amazing script!"
echo "Your current dir is: $(pwd)"
```

- *Exercise* : Write a script that when run on attu does the following:
  - clears the screen
  - displays the current date/time
  - Shows who is currently logged on & info about processor

Script example

```
#!/bin/bash
clear       # please do not use clear in your hw scripts!
echo "Today's date is $(date)"
echo

echo "These users are currently connected:"
w -h | sort
echo

echo "This is $(uname -s) on a $(uname -m) processor."
echo

echo "This is the uptime information:"
uptime
echo
echo "That's all folks!"
```

Comments

`# comment text`

- bash has only single-line comments; there is no /* ... */ equivalent

- Example:

```
#!/bin/bash
# Leonard's first script ever
# by Leonard Linux
echo "This is my amazing script!"
echo "The time is: $(date)"

# This is the part where I print my current directory
echo “Current dir is: $(pwd)"
```

Shell variables

- ***name=value*** *(declaration)*
  - must be written ***EXACTLY*** as shown; no spaces allowed
  - often given all-uppercase names by convention
  - once set, the variable is in scope until unset (within the current shell)

```
AGE=89
NAME="Mickey Mouse"
```

- ***$name*** *(usage)*

```
echo "$NAME is $AGE years old"
```

Produces:

```
Mickey Mouse is 89 years old
```

Common errors

- if you misspell a variable's name, a new variable is created

```
NAME=Ruth
...
Name=Rob                # oops; meant to change NAME
```

- if you use an undeclared variable, an empty value is used

```
echo "Welcome, $name"  # Welcome,
```

- when storing a multi-word string, must use quotes

```
NAME=Ruth Anderson          # Won't work
NAME="Ruth Anderson"        # $NAME is Ruth Anderson
```

More Errors…

- Using $ during assignment or reassignment
  - `$mystring=“Hi there”  # error`
  - `mystring2=“Hello”`
  - …
  - `$mystring2=“Goodbye”  # error`
- Forgetting echo to display a variable
  - `$name`
  - `echo $name`

Capture command output

***variable*=$(*command*)**

- captures the output of ***command*** into the given variable

- Simple Example:

```
FILE=$(ls *.txt)
echo $FILE
```

- More Complex Example:

```
FILE=$(ls -1 *.txt | sort | tail –n 1)
echo "Your last text file is: $FILE"
```

  - What if we use double quotes instead?

Double vs. Single quotes

Double quotes - Variable names are expanded & $() work

```
NAME="Bugs Bunny"
echo "Hi $NAME! Today is $(date)"
```

Produces:

```
Hi Bugs Bunny! Today is Tues Apr 25 13:37:45 PDT 2017
```

Single quotes – don't expand variables or execute commands in $()

```
echo 'Hi $NAME! Today is $(date)'
```

Produces:

```
Hi $NAME! Today is $(date)
```

Tricky Example:

- STAR=*
  - echo "You are a $STAR"
  - echo 'You are a $STAR'
  - echo You are a $STAR

Lesson: When referencing a variable, it is good practice to put it in double quotes.

Types and integers

- most variables are stored as strings
  - operations on variables are done as string operations, not numeric

- to instead perform integer operations:

```
x=42
y=15
let z="$x + $y"          # 57
```

- integer operators: `+ - * / %`
  - `bc` command can do more complex expressions

- if a non-numeric variable is used in numeric context, you'll get 0

Bash vs. Java

Java	Bash
`String s = "hello";`	`s=hello`
`System.out.println("s");`	`echo s`
`System.out.println(s);`	`echo $s`
`s = s + "s";  // "hellos"`	`s=${s}s`

`String s2 = "25";`
`String s3 = "42";`
`String s4 = s2 + s3;  // "2542"`
`int n = Integer.parseInt(s2)`
`  + Integer.parseInt(s3);  // 67`
	`s2=25`
	`s3=42`
	`s4=$s2$s3`
	`let n="$s2 + $s3"`

`x=3`

- `x` vs. `$x` vs. `"$x"` vs. `'$x'` vs. `\'$x\'` vs. `'x'`

Special variables

variable	description
$DISPLAY	where to display graphical X-windows output
$HOSTNAME	name of computer you are using
$HOME	your home directory
$PATH	list of directories holding commands to execute
$PS1	the shell's command prompt string
$PWD	your current directory
$SHELL	full path to your shell program
$USER	your user name

- these are automatically defined for you in every bash session

- *Exercise* : Change your `attu` prompt to look like this:

  `jimmy@mylaptop:$`

  - See `man bash` for more info (search on PROMPTING)

$PATH

- When you run a command, the shell looks for that program in all the directories defined in $PATH
- Useful to add commonly used programs to the $PATH

- Exercise: modify the $PATH so that we can directly run our shell script from anywhere
  - echo $PATH
  - PATH=$PATH:/homes/iws/rea

- What happens if we clear the $PATH variable?

set, unset, and export

shell command	description
`set`	With sets the value of a variable (not usually needed; can just use `x=3` syntax)
`unset`	deletes a variable and its value
`export`	sets a variable and makes it visible to any programs launched by this shell
`readonly`	sets a variable to be read-only (so that programs launched by this shell cannot change its value)

- typing `set` or `export` with no parameters lists all variables
- *Exercise*: set a local variable, and launch a new bash shell
  - Can the new shell see the variable?
  - Now go back and export and launch a shell again. Can you see it now?

Console I/O

shell command	description
`read`	reads value from console and stores it into a variable
`echo`	prints output to console
`printf`	prints complex formatted output to console

- variables read from console are stored as strings

- Example:

```
#!/bin/bash
read -p "What is your name? " name
read -p "How old are you? " age
printf "%10s is %4s years old" $name $age
```

Command-line arguments

variable	description
`$0`	name of this script
`$1, $2, $3, ...`	command-line arguments
`$#`	number of arguments
`$@`	array of all arguments

- slide20.sh:

```
#!/bin/bash
echo “Name of script is $0”
echo “Command line argument 1 is $1”
echo “there are $# command line arguments: $@”
```

  - `slide20.sh argument1 argument2 argument3`

for loops

```
for name in value1 value2 ... valueN; do
    commands
done
```

- Note the semi-colon after the values!
- the pattern after `in` can be:
  - a hard-coded set of values you write in the script
  - a set of file names produced as output from some command
  - command line arguments: `$@`
- *Exercise*: create a script that loops over every .txt file in the directory, renaming the file to .txt2

```
for file in *.txt; do
  mv $file ${file}2
done
```

for loop examples

```
for val in red blue green; do
    echo "val is: $val"
done

for val in $@; do
    echo "val is: $val"
done

for val in $(seq 4); do
    echo "val is: $val"
done
```

command	description
seq	outputs a sequence of numbers

Exercise

- Write a script `createhw.sh` that creates directories named hw1, hw2, … up to a maximum passed as a command-line argument.

```
$ ./createhw.sh 8
```

  - Copy `criteria.txt` into each assignment *i* as `criteria(2*i).txt`
  - Copy `script.sh` into each, and run it.
    - output: `Script running on hw3 with criteria6.txt ...`

Exercise solution

```
#!/bin/bash
# Creates directories for a given number of assignments.

for num in $(seq $1); do
        let CRITNUM="2 * $num"
        mkdir "hw$num"
        cp script.sh "hw$num/"
        cp criteria.txt "hw$num/criteria$CRITNUM.txt"
        echo "Created hw$num."
        cd "hw$num/"
        bash ./script.sh
        cd ..

done
```

Exit Status

- Every Linux command returns an integer code when it finishes, called its "**exit status**"
  - 0 usually* denotes success, or an OK exit status
  - Anything other than 0 (1 to 255) usually denotes an error
- You can return an exit status explicitly using the **exit** statement
- You can check the status of the last command executed in the variable **$?**

```
$ cat someFileThatDoesNotExist.txt
$ echo $?
1              # "Failure"
$ ls
$ echo $?
0              # "Success"
```

* One example exception: `diff` returns "0" for no differences, "1" if differences found, "2" for an error such as invalid filename argument

The test command

```
$ test 10 -lt 5
$ echo $?
1              # “False”, “Failure”
$ test 10 -gt 5
$ echo $?
0              # “True”, “Success”
```

- Another syntax for the test command:
  Don’t forget the space after [and before]

```
$ [ 10 -lt 5 ]
$ echo $?
1              # “False”, “Failure”
$ [ 10 -gt 5 ]
$ echo $?
0              # “True”, “Success”
```

test operators

comparison operator	description
=, !=, \<, \>	compares two **string** variables
-z, -n	tests if a string is empty (zero-length) or not empty (nonzero-length)
-lt, -le, -eq, -gt, -ge, -ne	compares **numbers**; equivalent to Java's <, <=, ==, >, >=, !=
-e, -f, -d	tests whether a given file or directory exists
-r, -w, -x	tests whether a file exists and is readable/writable/executable

```
if [ $USER = “husky14" ]; then
    echo ‘Woof! Go Huskies!’
fi

LOGINS=$(w -h | wc -l)
if [ $LOGINS -gt 10 ]; then
    echo ‘attu is very busy right now!’
fi
```

*Note: **man test** will show other operators.

if/else

```
if [ condition ]; then          # basic if
    commands
fi


if [ condition ]; then          # if / else if / else
    commands1
elif [ condition ]; then
    commands2
else
    commands3
fi
```

- The `[ ]` syntax is actually shorthand for a shell command called "***test***" (Try: "man test")
- there ***MUST*** be spaces as shown:
  `if` space `[` space ***condition*** space `]`
- include the semi-colon after `]` (or put "`then`" on the next line)

More if testing

compound comparison operators	description
`if [ expr1 -a expr2 ]; then ...` `if [ expr1 ] && [ expr2 ]; then ...`	and
`if [ expr1 -o expr2 ]; then ...` `if [ expr1 ] || [ expr2 ]; then ...`	or
`if [ ! expr ]; then ...`	not

```
# alert user if running >= 10 processes when
# attu is busy (>= 5 users logged in)
LOGINS=$(w -h | wc -l)
PROCESSES=$(ps -u $USER | wc -l)
if [ $LOGINS -ge 5 -a $PROCESSES -gt 10 ]; then
    echo "Quit hogging the server!"
fi
```

Common errors

- `[: -eq: unary operator expected`
  - you used an undefined variable in an `if` test
- `[: too many arguments`
  - you tried to use a variable with a large, complex value (such as multi-line output from a program) as though it were a simple int or string
- `let: syntax error: operand expected (error token is " ")`
  - you used an undefined variable in a `let` mathematical expression

safecopy Exercise

- Write a script called `safecopy` that will mimic the behavior of `cp -i` where *`from`* is a filename and *`to`* is a filename:

```
$ cp -i from.txt to.txt
Do you want to overwrite to.txt? (yes/no)
```

```
$ ./safecopy.sh from.txt to.txt
Do you want to overwrite to.txt? (yes/no)
```

safecopy Exercise Solution

```
#!/bin/bash

FROM=$1
TO=$2

if [ -e $TO ]; then
    read -p "Do you want to overwrite $TO?" ANSWER
    if  [ $ANSWER = "yes" ]; then
        cp $FROM $TO
    fi
else
    cp $FROM $TO
fi
```

while and until loops

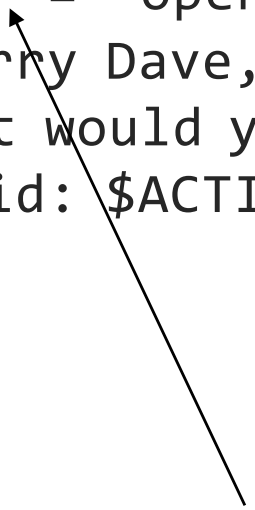
```
while [ condition ]; do   # go while condition is true
    commands
done


until [ condition ]; do   # go while condition is false
   commands
done
```

While exercise

- Prompt the user for what they would like to do. While their answer is "`open the pod bay doors`" tell them that you cannot do that and prompt for another action.

While Exercise solution

```
#!/bin/bash
# What would you like to do?
 read -p "What would you like me to do? " ACTION
 echo "You said: $ACTION"
 while [ "$ACTION" = "open the pod bay doors" ]; do
     echo "I'm sorry Dave, I'm afraid I can't do that."
     read -p "What would you like me to do? " ACTION
     echo "You said: $ACTION"
 done
 echo "Bye"
```



The quotes around “$ACTION” are important here, try removing them and see what happens.

select and case

- Bash Select statement:

```
PS3=prompt  # Special variable* for the select prompt
select choice in choices; do
        commands
        break # Break, otherwise endless loop
done
```

- Bash Case statement:

```
case EXPRESSION in
    CASE1) COMMAND-LIST;;
    CASE2) COMMAND-LIST;;
    ...
    CASEN) COMMAND-LIST;;
esac
```

*see lecture 5

Select Example

```
PS3="What is your favorite food? " # Goes with the select stmt

echo "Welcome to the select example!"
echo "It prints out a list of choices"
echo "but does nothing interesting with the answer."

select CHOICE in "pizza" "sushi" "oatmeal" "broccoli"; do
    echo "You picked $CHOICE"
    break
done

echo “For the select statement, you pick a number as your choice."
```

Case Example

```
echo "Welcome to the case example!"
echo "Without a select statement, you must get the spelling/case exact."
read -p "What format do you prefer? (tape/cd/mp3/lp) " FORMAT
echo "You said $FORMAT"

case "$FORMAT" in
    "tape") echo "no random access!";;
    "cd") echo "old school";;
    "mp3") echo "how modern";;
    "lp") echo "total retro";;
esac
```

select/case Exercise

- Have the user select their favorite kind of music, and output a message based on their choice

select/case Exercise Solution

```
PS3="What is your favorite kind of music? "
select CHOICE in "rock" "pop" "dance" "reggae"; do
    case "$CHOICE" in
        "rock") echo "Rock on, dude.";;
        "pop") echo "Top 100 is called that for a reason.";;
        "dance") echo "Let's lay down the Persian!";;
        "reggae") echo "Takin' it easy...";;
        * ) echo "come on...you  gotta like something!";;
    esac
    break
done
```

Arrays

```
name=(element1 element2 ... elementN)

name[index]=value          # set an element
$name                      # get first element
${name[index]}             # get an element
${name[*]}                 # elements sep.by spaces
${#name[*]}                # array's length
```

- arrays don't have a fixed length; they can grow as necessary
- if you go out of bounds, shell will silently give you an empty string
  - you don't need to use arrays in assignments in this course

Functions

```
function name() {          # declaration
    commands               # ()'s are optional
}

name                       # call
```

- functions are called simply by writing their name (no parens)
- parameters can be passed and accessed as $1, $2, etc.
  - you don't need to use functions in assignments in this course