CSE 390a
Lecture 5

Intro to shell scripting

slides created by Marty Stepp, modified by Jessica Miller
http://www.cs.washington.edu/390a/
Lecture summary

- basic script syntax and running scripts
- shell variables and types
- control statements: the for loop
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:

```bash
#!/bin/bash
rm output*.txt
ls -l
```
Running a shell script

• by making it executable (most common; recommended):
  
  ```bash
  chmod u+x myscript.sh
  ./myscript.sh
  ```

• by launching a new shell:
  
  ```bash
  bash myscript.sh
  ```

• by running it within the current shell:
  
  ```bash
  source myscript.sh
  ```

  ▪ advantage: any variables defined by the script remain in this shell
  (seen later)
**echo**

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>echo</td>
<td>produces its parameter(s) as output (the <code>println</code> of shell scripting)</td>
</tr>
<tr>
<td></td>
<td><code>-n</code> flag to remove newline (<code>print vs </code>println`)</td>
</tr>
</tbody>
</table>

- **Example:** A script that prints your home directory.

```bash
#!/bin/bash
echo "This is my amazing script!"
echo "Your home dir is: `pwd`"
```

- **Exercise:** Write a script that when run on attu does the following:
  - clears the screen
  - displays the date/time: Today’s date is Tue Apr 26 10:44:18 PDT 2011
  - shows me an ASCII cow welcoming my user name
#!/bin/bash

clear
echo "Today's date is `date`"
echo
~stepp/cowsay `whoami`

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`"

echo "Home 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)

  NUMFRIENDS=2445
  NAME="Guess who"

• **$name**
  
  (usage)

  echo "**$NAME** has **$NUMFRIENDS** FB friends"
  Guess who has 2445 FB friends
Common errors

• if you misspell a variable's name, a new variable is created
  NAME=Marty
  ...
  Name=Daniel  # 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=Marty Stepp  # $NAME is Marty
  NAME="Marty Stepp"  # $NAME is Marty Stepp
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

\[ \text{variable} = `\text{command}` \]

- captures the output of \texttt{command} into the given variable

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

- What if we leave off the last backtick?
- What if we use quotes instead?
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
  \text{let } z="x + y" \quad \# \ 57

- integer operators: + - * / %
  - \text{bc} command can do more complex expressions

- if a non-numeric variable is used in numeric context, you'll get 0
## Bash vs. Java

<table>
<thead>
<tr>
<th>Java</th>
<th>Bash</th>
</tr>
</thead>
<tbody>
<tr>
<td>String s = &quot;hello&quot;;</td>
<td>s=hello</td>
</tr>
<tr>
<td>System.out.println(&quot;s&quot;);</td>
<td>echo s</td>
</tr>
<tr>
<td>System.out.println(s);</td>
<td>echo $s</td>
</tr>
<tr>
<td>s = s + &quot;s&quot;;</td>
<td>s=${s}s</td>
</tr>
<tr>
<td>String s2 = &quot;25&quot;;</td>
<td>s2=25</td>
</tr>
<tr>
<td>String s3 = &quot;42&quot;;</td>
<td>s3=42</td>
</tr>
<tr>
<td>String s4 = s2 + s3;</td>
<td>s4=${s2}${s3}</td>
</tr>
<tr>
<td>int n = Integer.parseInt(s2) + Integer.parseInt(s3); // 67</td>
<td>let n=&quot;${s2} + ${s3}&quot;</td>
</tr>
</tbody>
</table>

### x=3
- x vs. $x vs. "$x" vs. '\$x' vs. x vs. '\$x' vs. 'x'
Special variables

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

- these are automatically defined for you in every bash session

- Exercise: Change your attu prompt to look like this:
  jimmy@mylaptop:$
$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/jessica

• What happens if we clear the $PATH variable?
set, unset, and export

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

- 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. Result?
## Console I/O

<table>
<thead>
<tr>
<th>shell command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>reads value from console and stores it into a variable</td>
</tr>
<tr>
<td>echo</td>
<td>prints output to console</td>
</tr>
<tr>
<td>printf</td>
<td>prints complex formatted output to console</td>
</tr>
</tbody>
</table>

- variables read from console are stored as strings

### Example:

```bash
#!/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**

<table>
<thead>
<tr>
<th>variable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>name of this script</td>
</tr>
<tr>
<td>$1, $2, $3, ...</td>
<td>command-line arguments</td>
</tr>
<tr>
<td>$#</td>
<td>number of arguments</td>
</tr>
<tr>
<td>$@</td>
<td>array of all arguments</td>
</tr>
</tbody>
</table>

- **Example.sh:**
  
  ```bash
  #!/bin/bash
  echo "Name of script is $0"
  echo "Command line argument 1 is $1"
  echo "there are $# command line arguments: $@
  ```

  ```bash
  Example.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
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 ...

  ▪ The following command may be helpful:

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>outputs a sequence of numbers</td>
</tr>
</tbody>
</table>
#!/bin/bash

# Creates directories for a given number of assignments.

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