TOPIC 15: BASH SCRIPTING

variables, read, command execution, $?, ``, $(), let, bc, conditionals, loops, select
script.sh:

#!/bin/bash

echo “Hello World”

UNIX> chmod u+x script.sh
UNIX> ./script.sh
Hello World!
UNIX>
first BASH script

script.sh:

```bash
#!/bin/bash
```

- first line “#!/bin/bash” specifies which shell program to use

```bash
#echo "Hello World"
```

- script contents are the commands we’ve learned!
  - (Plus some other stuff…)

- (Plus some other stuff…)
variables can be declared and used in BASH scripts

to declare variables:
  - x=10
  - pi=3.14159
  - str="this is a string"

Note the lack of declared types

Note the lack of spaces
BASH variables

- to use a `variable`, prepend it with a `$`:

  $x
  $\pi$
  $str$
BASH variables

script.sh:

```
#!/bin/bash
x=10
pi=3.14159
str="This is a string"

echo x pi str
echo $x $pi $str
```

UNIX> ./script.sh

x pi str
10 3.14159 This is a string
**BASH read**

*read* command to get *stdin* from the user

**script.sh:**

```bash
#!/bin/bash
echo "Enter x"
read x
echo You entered $x
```

**UNIX> ./script.sh**

Enter x

8 <ENTER>

You entered 8

UNIX> ./script.sh

Enter x

Hello There! <ENTER>

You entered Hello There!
As mentioned before, BASH scripts execute commands as if entered on the command line

```
#!/bin/bash
seq 10 | wc -l
echo Hello World!
```

```
UNIX> seq 10 | wc -l
10
UNIX> echo Hello World!
Hello World!
UNIX> ./script.sh
10
Hello World!
```
Bash `?`

- `?` stores the previous command’s return value
  - 0 for success
  - !0 for error

```
UNIX> ls /dev/null
UNIX> echo ?
0
```

```
UNIX> ls no_file.txt
ls: no_file.txt: No such file or directory
UNIX> echo ?
2
```
BASH $? – zero for OK, non-zero for errors

UNIX> echo A | grep B
UNIX> echo $?  
1

UNIX> echo A | grep A
A
UNIX> echo $?  
0

UNIX> g++ errors.cpp
... (compiler errors)...
UNIX> echo $?  
1

UNIX> g++ main.cpp
UNIX> echo $?  
0
a command placed in **backticks** will be evaluated before used...

UNIX> echo ls
ls

UNIX> echo `ls`
script.sh
BASH $(())

- $(command) works the same way as backticks

UNIX> echo ls
ls

UNIX> echo $( ls )
script.sh
• *let* command can be used for integer arithmetic

**script.sh:**

```bash
#!/bin/bash
x=2
y=2
let "z = $x + $y"
echo $z
```

**UNIX> ./script.sh**

4

**UNIX>**

› Note the double quotes
  - Double quotes allow for whitespace between operators and variables
**BASH let**

- *let* only calculates integers (note integer division!)

**script.sh:**

```bash
#!/bin/bash
x=1
y=2
let z=$x/$y
echo $z
```

**UNIX> ./script.sh**

0

**UNIX>**

No spaces between variables and operators!
see lecture notes for list of valid `let` operators

e.g.,:

+, −, =, *, /, %, **, ++, etc.
BC is a command line calculator
Use BC to calculate decimals in your scripts

UNIX> echo 2 + 2 | bc
4

UNIX> echo 1 / 2 | bc
0

- Zero!!?!? We must “scale” the input..
to calculate floating points, use the “scale” operation for \texttt{bc}

\begin{verbatim}
UNIX> echo 1 / 2 | bc
0

UNIX> echo "scale=3; 1 / 2" | bc
.500
\end{verbatim}

\textit{scale} specifies the floating point precision (e.g., 3)
• Use `bc` in a BASH script:

```bash
#!/bin/bash

pi=3.14159
r=2.5

area=$( echo "scale=3; \$pi*\$r^2" | bc )

echo \$area
```

```
UNIX> ./script.sh
19.63493
```
Modify the flow of execution with if statement(s)

Basic syntax:

```
if [ condition ]; then
  command
  ...
fi
```
The arithmetic logic operators used in BASH scripts are different than other languages*

<table>
<thead>
<tr>
<th>logical operator</th>
<th>BASH operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>-eq</td>
</tr>
<tr>
<td>!=</td>
<td>-ne</td>
</tr>
<tr>
<td>&gt;</td>
<td>-gt</td>
</tr>
<tr>
<td>&lt;</td>
<td>-lt</td>
</tr>
<tr>
<td>&gt;=</td>
<td>-ge</td>
</tr>
<tr>
<td>&lt;=</td>
<td>-le</td>
</tr>
</tbody>
</table>

*see lecture notes for string and file logic operators
**BASH conditionals**

- complex Boolean expressions (logical OR, AND, NOT)

<table>
<thead>
<tr>
<th>logical operation</th>
<th>BASH operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>-a</td>
</tr>
<tr>
<td>OR</td>
<td>-o</td>
</tr>
<tr>
<td>NOT</td>
<td>!</td>
</tr>
</tbody>
</table>
BASH conditionals

script.sh:

#! /bin/bash
read x
if [ $x -eq 10 ]; then
echo “x equals 10”
fi

UNIX> ./script.sh
5 <ENTER>
UNIX> ./script.sh
10 <ENTER>
x equals 10
BASH conditionals

script.sh:

#!/bin/bash
read x
if [ $x -lt 0 -o $x -gt 100 ]; then
    echo bad input: $x
fi

UNIX> ./script.sh
50 <ENTER>
UNIX> ./script.sh
-10 <ENTER>
bad input: -10
UNIX> ./script.sh
101 <ENTER>
bad input: 101
### BASH if / else ; if / else if / else

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Command(s)</th>
<th>Conditions</th>
<th>Command(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if [ condition ]; then</td>
<td>command(s)</td>
<td>if [ condition ]; then</td>
<td>command(s)</td>
</tr>
<tr>
<td>else</td>
<td>command(s)</td>
<td>elif [ condition 2]; then</td>
<td>command(s)</td>
</tr>
<tr>
<td>fi</td>
<td></td>
<td>else</td>
<td>command(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fi</td>
<td></td>
</tr>
</tbody>
</table>
script.sh:

#!/bin/bash
read x;
if [ $x -lt 10 ]; then
    echo "x < 10"
elif [ $x -gt 10 ]; then
    echo "x > 10"
else
    echo "x == 10"
fi

UNIX> echo 9 | ./script.sh
x < 10

UNIX> echo 11 | ./script.sh
x > 10

UNIX> echo 10 | ./script.sh
x == 10
• repeat blocks of code with **for** and **while** loops

  ‣ **for** loops:

  ```bash
  for var in 1 2 3 4; do
    command(s)
    ...
  done
  ```

  ‣ **while** loops:

  ```bash
  while [ condition ]; do
    command(s)
    ...
  done
  ```
**BASH for loops**

- `for` loops expect literal values to iterate over
- These values can be integers, double, strings, etc.

**script.sh:**
```bash
#!/bin/bash
for x in 1 2 3 4; do
echo $x
done
```

**UNIX> ./script.sh**
```
1
2
3
4
```
script.sh:

```bash
#!/bin/bash
for x in Dog Cat Goat; do
    echo $x
done
```

UNIX> ./script.sh

Dog
Cat
Goat

‣ `for` loops can work with strings...
but.. what if we want to iterate over 10000 numbers?

easy! use `backticks`!

```
#!/bin/bash
for x in `seq 10000`; do
    echo $x
done
```

UNIX> ./script.sh | wc -l
10000
Using `backticks`, it’s easy to, say, loop through all files in the directory:

```bash
#!/bin/bash
for x in `ls`; do
touch $x
done
```

```bash
UNIX> ls -l | awk {'print $8'}
17:29
```

```bash
UNIX> ./script.sh
UNIX> ls -l | awk {'print $8'}
```

17:32
while loops have the following syntax:

```
while [ condition ]; do
    command(s)
...
done
```

- The condition is a Boolean expression
- The while loop iterates *while* the condition is true
Use a `while` loop to iterate 4 times:

```bash
#!/bin/bash
x=0
while [ $x -lt 4 ]; do
    echo $x
    let "x++"
done
```

UNIX> ./script.sh
0
1
2
3
Use a `while` loop to wait for some input

**script.sh:**

```bash
#!/bin/bash
x=-1;
while [ $x -lt 0 -o $x -gt 100 ]; do
echo "Enter x: 0 < x < 100: ";
read x;
done
echo You entered $x
```

**UNIX> ./script.sh**
Enter x: 0 < x < 100:
-50 <ENTER>
Enter x: 0 < x < 100:
150 <ENTER>
Enter x: 0 < x < 100:
50 <ENTER>
You entered 50
UNIX>
select allows easy menu generation

select is a looping mechanism (note the break)

script.sh:
```
#!/bin/bash
select i in First Second Third; do
  break;
done
echo You selected $i
```

UNIX> ./script.sh
1) First
2) Second
3) Third
#? 2 <ENTER>
You selected Second
E.g., use `select` to choose files at runtime...

```bash
#!/bin/bash
select file in `ls`; do
    break;
done
```

```
UNIX> touch f1 f2 f3
UNIX> ./script.sh
1) f1  3) f3
2) f2  4) script.sh
#? 3 <ENTER>
You selected f3
```
We have only scratched the surface...

BASH scripts can have functions, arrays, command line arguments, switch statements, etc.

Write BASH scripts to automate stuff. E.g.,
- grading
- simulations
- simple daemon processes
- etc.
Assignment 15

- [http://eecs.mines.edu/Courses/csci274/Assignments/15_script.html](http://eecs.mines.edu/Courses/csci274/Assignments/15_script.html)

- Write BASH scripts to:
  - auto grade student “submissions”
  - simulate solar panel efficiency
  - test runtimes of various sorting algorithms