

Unix Shell Programming

*Computer Science Department
CS2211a: Software Tools & Systems Programming
Fall 2013
Instructor: Mahmoud R. El-Sakka
Office: MC-419
Email: elsakka@csd.uwo.ca
Phone: 519-661-2111 x86996*

1

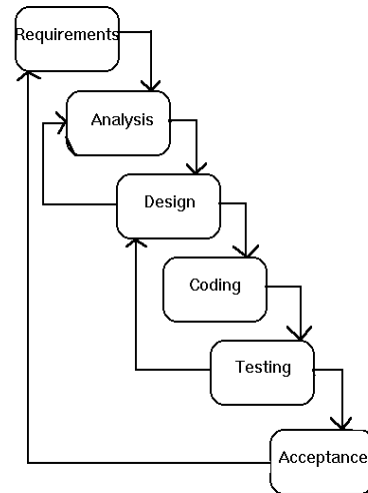
Topic 09: Unix Shell Programming

Shell Scripts

- A shell script is a text file that contains *Unix commands* and *flow control commands* to be executed by the shell
- It can be read and understood *by humans* and *by machines*
- Commands are executed
 - ☐ in order as they appear in the file or
 - ☐ in the flow determined by control statements
- Different shells have different control structures

Shell Scripts

- Like high-level source files, a programmer creates shell scripts with a text editor.
- Unlike high-level language programs, shell scripts do not have to be converted into machine language by a compiler.
 - Unix shell acts as an interpreter when reading script files.
- The shell *script development cycle* is exactly the same as in the high-level programming case.



Shell Scripts

- Unix must be informed which shell will be used to run the script
 - Method 1:
 - Calling the shell followed by the script file name
 - There is *no need* to set the permission of a script to executable
 - For example: `sh my_Bourne_shell_script`
`tcsh my_C_shell_script`
 - Method 2:
 - Invoking the shell as the *first working line* in the script file (using `#!`)
 - In this case, the permission of a script *must* be set to executable
 - For example: `#!/bin/sh`, `#!/bin/csh`, `#!/bin/bash`, or `#!/bin/ksh`
 - If `#!` is not used to invoke the shell at the beginning of an executable shell script, a default *shell* will be used
 - If both methods are presented, the first one has a precedence
 - The second method is most appropriate; *Why?*
- We will write shell scripts with the *Bourne shell* (`sh`)

Shell Scripts

- Why shell scripts?
 - To avoid repetition
 - If you do a sequence of steps with standard Unix commands over and over, why don't we do them all using just one command?
 - To automate difficult tasks
 - Many commands have difficult options that you don't want to remember or figure out every time

A Simple Example

- `tr abcdefghijklmnopqrstuvwxyz \thequickbrownfxjimpsvalzydg < file1 > file2`
 - *encrypts* file1 into file2
- We can save such commands into two shell script files
 - *myencrypt*

```
#!/bin/sh
tr abcdefghijklmnopqrstuvwxyz thequickbrownfxjimpsvalzydg
```
 - *mydecrypt*

```
#!/bin/sh
tr thequickbrownfxjimpsvalzydg abcdefghijklmnopqrstuvwxyz
```

A Simple Example

- `chmod` of *myencrypt* and *mydecrypt* files to make them executable
`chmod u+x myencrypt mydecrypt`
- Run them as normal commands
`./myencrypt < file1 > file2`
`./mydecrypt < file2 > file3`
`diff file1 file3`
- *Homework*: try to encrypt/decrypt text using the above commands

Bourne Shell Variables

- *Bourne shell* variables are different from *csh* and *tcsh* variables
- Examples in sh:
 - HA=\$1
 - BUY="House in a nice place"
 - export BUY
 - PATH=\$PATH:\$HOME/bin

no space
around
=

export makes BUY an
environment variable

Assigning Command Output to a Variable

- Using *backquotes*, we can assign the output of a command to a variable

```
#!/bin/sh
files=`ls`
echo $files
```

Using expr for Calculations

```
#!/bin/sh
value=`expr 12345 + 54321`
echo $value
```

expr evaluates
expressions

Must have spaces
surrounding each operator

- Assume that you execute the following commands at the Unix prompt


```
count=5
count=`expr $count + 1`
echo $count
6
echo "$count"
6
echo '$count'
$count
echo \count
$count
```
- Operators include: +, -, *, /, %

Shell Script Example

```
#!/bin/sh
clear
echo "Hello, $USER"
echo

echo "Today's date is `date`."
echo

echo "These users are currently connected:"
w | cut -d " " -f 1 | grep -vi user | sort -u
echo

echo "This is `uname -s` running on a `uname -m` processor."
echo

echo "This is the uptime information:"
uptime
echo
```

Shell Script Example

- When executing the previous script, you will get something like this:
Hello, elsakka

Today's date is Thu Sep 27 11:52:37 EDT 2012.

These users are currently connected:

```
andrews
bruce
elsakka
jgodisar
lmoyo
mlocke2
mschuber
txijierf
```

This is SunOS running on a sun4u processor.

This is the uptime information:
11:52am up 16 day(s), 15:43, 8 users, load average: 0.06, 0.07, 0.05

Control Statements

- Without control statements, execution within a shell script flows sequentially from one statement to the next
- In general, control statements govern the flow of execution in any programming language
- The most common types of control statements are
 - loop statements: `for`, `while`, and `until`
 - conditionals: `if/then/else`, and `case`

for Loop

- `for` loops allow the repetition of a command for a specific set of values

- Syntax:

```
for variable in value1 value2 ...
do
    command_set
done
```

Needs end-of-line or
a ; separator

Does not Need
end-of-line separator

Needs end-of-line or ; separator

- `command_set` is executed with each value of `variable` in sequence (i.e., `value1`, `value2`, ...)

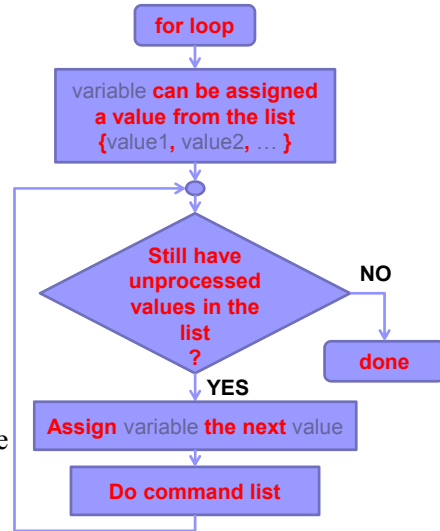
for Loop

- for loops allow the repetition of a command for a specific set of values

- Syntax:

```
for variable in value1 value2 ...
do
    command_set
done
```

- command_set is executed with each value of variable in sequence (i.e., value1, value2, ...)



for Loop

```
#!/bin/sh
# timestable - print out a multiplication table
for i in 1 2 3
do
    for j in 1 2 3
    do
        value=`expr $i \* $j`
        echo -n "$value "
    done
    echo
done

timestable
1 2 3
2 4 6
3 6 9
```

This line is a comment.

Why did we use "*", and not just "*"?

Why did we use double quotations around \$value?

-n means: do not add newline at the end of the output

for Loop

To debug your program use **-x** when you invoke the shell

```
#!/bin/sh
# timestable_two - only 2 by 2
for i in 1 2
do
    for j in 1 2
    do
        value=`expr $i \* $j`
        echo -n "$value "
    done
    echo
done
```

The **-x** can be put here as well

```
sh -x timestable_two
+ expr 1 * 1
value=1
+ echo -n '1 '
1 + expr 1 * 2
value=2
+ echo -n '2 '
2 + echo
+ expr 2 * 1
value=2
+ echo -n '2 '
2 + expr 2 * 2
value=4
+ echo -n '4 '
4 + echo
```

for Loop

To debug your program you can also use **-xv** when you invoke the shell

```
#!/bin/sh
# timestable_two - only 2 by 2
for i in 1 2
do
    for j in 1 2
    do
        value=`expr $i \* $j`
        echo -n "$value "
    done
    echo
done
```

```
sh -xv timestable_two
#!/bin/sh
# timestable_two - only 2 by 2
for i in 1 2
do
    for j in 1 2
    do
        value=`expr $i \* $j`
        echo -n "$value "
    done
done
echo
+ expr 1 * 1
value=1
+ echo -n '1 '
```

```
1 + expr 1 * 2
value=2
+ echo -n '2 '
2 + echo
+ expr 2 * 1
value=2
+ echo -n '2 '
2 + expr 2 * 2
value=4
+ echo -n '4 '
4 + echo
```

- The **-xv** can be also put at the **#!/bin/sh** as well

for Loop

```
#!/bin/sh
# To look at files in a directory
files=`ls`
for filename in $files
do
    echo $filename
done
```



```
obelix[41]% look_at_files
data
look_at_files
readme.txt
tmp
obelix[42]%
```

for Loop

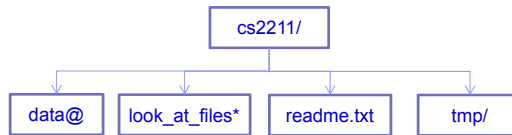
```
#!/bin/sh
# To look at files in a directory
files=`ls`
echo $files
for filename in $files
do
    echo $filename
done
```



```
obelix[43]% look_at_files
data look_at_files readme.txt tmp
data
look_at_files
readme.txt
tmp
obelix[44]%
```

for Loop

```
#!/bin/sh
# To look at files in a directory
files=`ls`
for filename in $files
do
    wc -c $filename
done
```



```
obelix[45]% look_at_files
0 data
104 look_at_files
290 readme.txt
0 tmp
obelix[46]%
```

for Loop

```
#!/bin/sh
# file-poke - tell us stuff about files
files=`ls`
for i in $files
do
    echo "---- $i ----"
    grep $i $i
    echo "-----"
done
```



- Find lines in a file (in current directory) that contain its filename
- *Homework* : use **-xv** to trace the above script on a small directory

for Loop

```
#!/bin/sh
# file-poke - tell us stuff about files
```

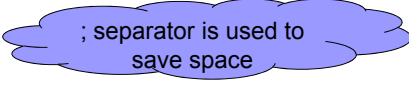
```
for i in *; do
```

```
    echo "---- $i ----"
```

```
    grep $i $i
```

```
    echo "-----"
```

```
done
```



; separator is used to
save space

- Same as previous slide, only a little more condensed