Linux Shell Part 2

Shell Script

A text file containing shell commands and/or shell language statements is a shell script.

A comment in a shell script is indicated by beginning the line with a "#".

In a shell script, a **shebang** interpreter directive tells Linux which command interpreter to use. Example shebangs:

```
#!/bin/bash
#!/bin/tcsh
#!/usr/bin/perl
```

Alternatively, instead of specifying the location of the interpreter, you can use /usr/bin/env to search the user's PATH for the correct interpreter. For example:

```
#!/usr/bin/env bash
```

The above will defer to PATH to find the location of bash. This can be useful when you aren't certain of the interpreter's location or if the user prefers to use a different binary than the installed one.

When debugging, it's useful to use the **`set -x`** command at the beginning of your script (after the shebang). This makes all commands print to stdout as they are executed. **DO NOT turn in an assignment with `set -x` present.**

Notes about UTSA:

- Login shell is now **bash** (used to be tcsh)
- When you launch a shell script which doesn't contain a shebang, the launched shell is neither bash nor tcsh; instead, it is **dash**.

```
$ cat >whoson
#!/bin/bash
# Shows the date and who is on
date
echo "who is on?"
who
CTRL-D
$
```

Shell Variables

Shell variables are similar to variables in other languages since they represent other values. The variable names must begin with a letter or underscore, but can contain letters, numbers, and underscores.

The value of a variable is *referenced* by preceding it with a $. (Note that we will see how numeric calculations might not use $ references.)

```
Valid variable names:
      line, name1, name2, first_name
Invalid variable names:
      first-name, last.name, 1char

# Variable references
$ cat >echoName
echo "Number of parameters is $#"
```

The **parameters** to a shell script are referenced by a "$" followed by a positional number. The name of the command is $0. $1 is the first parameter, $2 is the second parameter, and so on. The number of parameters is $#. $@ represents the *list* version of the parameters.

Variable names are case sensitive.

```
echo "Full name is $1 $2"
name=$1
echo "First name is $name"
CTRL-D

$ chmod u+x echoName
$ ./echoName bob wire
Number of parameters is 2
Full Name is bob wire
First name is bob
```

Setting Variable Values

Shell scripts provide assignment statements which can give shell variables values during the execution of the shell.

bash/dash:

`targetVariable=value` (no spaces around "=")

We will learn about the scope of variables below.

```
# assign some variables using a bash script
$ cat > assignVar
#!/bin/bash
first=ray
last=king
echo $first $last
CTRL-D
$ chmod u+x assignVar
$ ./assignVar
ray king

# what are the values after the shell script executed?
$ echo $first $last
```

Assigning From Math Expressions in BASH

Four forms:

```
let targetVariable=expression
let targetVariable=$((expression))
targetVariable=$((expression))
((targetVariable=expression))
```

Sam's note: $((…)) is arithmetic *expansion* of an arithmetic *expression* (no enclosed = signs); ((…)) and let… are equivalent; indicate arithmetic *evaluation* of an expression

Note that the double parentheses allow spacing around operators within the double parentheses.

The **let** command causes the variables to undergo expansion.

```
# create a shell script for simple math taking two parameters
$ vi simpMath
#!/bin/bash
let sum=$1+$2
let product=$(($1 * $2))
diff=$(($1 - $2))
((quotient = $1 / $2))
echo "sum is $sum, product is $product"
echo "diff is $diff, quotient is $quotient"

$ chmod u+x simpMath
$ ./simpMath 3 2
sum is 5, product is 6
diff is 1, quotient is 1
```

Floating Point

Note that floating point values do *not* work in either bash or tcsh. If necessary, pipe the expression into a utility such as **bc** (basic calculator).

```
# attempt to handle floating point with bash
$ echo $((3 / 2))
1

# attempt to handle floating point with bc
$ echo "3 / 2" | bc -l
1.50000000000000000000
```

Exercise

```
# what is the result of the following script
$ cat > massign
 #!/bin/bash
 one=two
 two=one
 let $two=2
 let one=2
 echo "one is $one"
 echo "two is $two"
CTRL-D
$ chmod u+x massign
$ ./massign
??
```

Some Important Variables

There are several important environment variables (global variables) for your login session:

PATH	list of directories to search to find commands. This is initially established by a Systems Administrator having root privileges.
HOME	this is where your home directory exists. The shell uses this for the value of ~.
PWD	this is the current directory.

Use echo $variable to show the value of a particular variable.

```
# show the contents of PATH
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
:/sbin:/bin:/usr/local/faculty/bin:.:/home/ssilvestro/bin
```

Showing Env Variables

You can see all of your current environment variables by using the **printenv** command.

```
# show all the environment variables
$ printenv | more
USER=ssilvestro
LOGNAME=ssilvestro
HOME=/home/ssilvestro
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/local/faculty/bin:.:/home/ssilvestro /bin
```

```
MAIL=/var/mail/ssilvestro
SHELL=/bin/bash
--More--
```

Setting PATH
You can set your path to include your bin. **Make certain your ~/bin exists.** If not, mkdir it.

bash:

- To have it set each time you start bash, change your
  `~/.bashrc`
- To change PATH to search ~/bin after the other directories, use
  `PATH=$PATH:$HOME/bin`

```
# It is very common to specify your own "bin" directory be added
# to PATH in your .bash_profile (for bash) or .cshrc (for
csh/tcsh)
# see which you have.  The "a" switch shows all files including
# hidden files.
# ls -al ~/.*rc
-rwx--x--x 1 ssilvestro faculty   441 Jul 14  2015 /home/ssilvestro
/.cshrc
-rw-r--r-- 1 ssilvestro faculty    43 May 31  2016 /home/ssilvestro
/.dmrc
-rw------- 1 ssilvestro faculty    97 Oct 17  2016 /home/ssilvestro
/.vimrc

# create your ~/bin
$ mkdir ~/bin



# If you want to add your own bin directory (~/bin) to
# .bash_profile, make certain it exists, and add this to the
# .bash_profile (be careful) using vi
```

Scope of Shell Variables
By default, shell variables are only known to the script and do not propagate to any script invoked from that script.

```
# Create the following scripts
$ cat >extest1
echo "$0 0: $cheese"
cheese=american
echo "$0 1: $cheese"
./subtest1
echo "$0 2: $cheese"
CTRL-D

$ cat >subtest1
echo "$0 0: $cheese"
cheese=swiss
echo "$0 1: $cheese"
CTRL-D

$ chmod u+x *test1
# execute extest1
$ ./extest1
./extest1 0:
./extest1 1: american
```

	`./subtest1 0:`
	`./subtest1 1: swiss`
	`./extest1 2: american`
	`# notice that cheese did not propagate to subtest1 and the`
	`# change to cheese in subtest1 is not known in extest1.`

Exporting variables
You can propagate shell variables to invoked shell scripts by exporting the variables. Exporting a variable makes it an **environment variable**, which is a global variable for shells. The variable will be available to any sub-shells
	`# copy extest1 to extest2 and add an export`
	`$ cp extest1 extest2`
	`$ vi extest2`
	`echo "$0 0: $cheese"`
	`cheese=american`
	`export cheese`
	`echo "$0 1: $cheese"`
	`./subtest1`
	`echo "$0 2: $cheese"`
	`$`
	`# What do you expect to happen?`
	`# The value "american" should be known to subtest1.`
	`# Will "swiss" be propagated back to extest2?`
	
	`$ ./extest2`
	`./extest2 0:`
	`./extest2 1: american`
	`./subtest1 0: american`
	`./subtest1 1: swiss`
	`./extest2 2: american`
	
	`# What happens if we run extest1 again?`
	`$ ./extest1`
	`./extest1 0:`
	`./extest1 1: american`
	`./subtest1 0:`
	`./subtest1 1: swiss`
	`./extest1 2: american`

How do I get an invoked shell script to set variables in the current shell?
Depending on the shell, you can either use "**.**" or **source** when invoking a shell script. This causes the invoked shell script to execute as part of the current process instead of a new process. Changes to variables in the invoked script will affect the current shell.
	`# create another version of extest2`
	`$ cp extest2 extest3`
	`$ vi extest3`
	`echo "$0 0: $cheese"`
	`cheese=american`
	`export cheese`
	`echo "$0 1: $cheese"`
	`source ./subtest1`
	`echo "$0 2: $cheese"`

```
# Invoke extest3
$ ./extest3
./extest3 0:
./extest3 1: american
./extest3 0: american
./extest3 1: swiss
./extest3 2: ??

What changed?
??
```

Exit Status
In addition to commands writing output to stdout, commands have an **exit status**:

0	success
non-zero	command failed

This can be tested to see whether something that was invoked actually worked.

Your shell commands can return a failure by executing:
exit *n*
To show a failure, the value of *n* will typically be 1 for most scripts.

You can get the last exit status by accessing the special value $?.

```
# attempt to show an unknown file
$ cat xxx
cat:  xxx: No such file or directory
$ echo $?
1
```

Job Sequences
Within shell scripts, we can specify job sequences, which are an easy way to link two commands based on the execution status.

cmd1 args && *cmd2 args*   # logical and
cmd2 only executes if *cmd1* returns 0.
cmd1 args || *cmd2 args*   # logical or
cmd2 only executes if *cmd1* returns non-zero.
cmd1 args ; *cmd2 args*
cmd2 executes regardless of *cmd1's* return value.

The approach of using && and || is like short circuiting in many languages.

If it is necessary for *cmd2* to be multiple commands, surround them in parentheses.

```
$ mkdir ~/cs3423/Jobs
# Create a backup directory, if it fails show an error.
# The name for the backup directory will be "Backup" followed by
# the current date.
# Note that the \ is used to continue the command. The question mark
# is a shell prompt for the continuation.
$ mkdir "Backup`date +%Y%m%d`" || \
? echo "creation of Backup directory failed"

# Execute it again
$ mkdir "Backup`date +%Y%m%d`" || \
? echo "creation of Backup directory failed"
mkdir: cannot create directory Backup20170713: File exists
creation of Backup directory failed

# Create a Backup directory. If it is successful,
# cp the contents of a folder to it.  cp -r will
# recursively create/copy any subdirectories,
$ mkdir "Backup`date +%Y%m%d`" && cp -r ~/cs2123/* "Backup`date +%Y%m%d`"
mkdir: cannot create directory Backup20170713: File exists
# Remove the backup directory (whatever its name is)
$ rmdir Backup20170713
```

```
# make the backup copy only if creating the directory is ok
$ mkdir "Backup`date +%Y%m%d`" && cp -r ~/cs2123/* "Backup`date +%Y%m%d`"

# Create the backup directory and copy the files.  If either of those
# failed, show an error message.
$ ( mkdir "Backup`date +%Y%m%d`" && \
? cp -r ~/cs2123/* "Backup`date +%Y%m%d`" ) || echo "backup failed";
mkdir: cannot create directory Backup20170713: File exists
backup failed

# Remove the backup directory (whatever its name is) and try that again
...
```

Reading input from stdin (works in bash and dash)

The **read** built-in command reads from stdin. Syntax:

read *variable*	reads the input into the specified variable until linefeed
read *var1 var2*	reads the first word into *var1*, the second word into *var2*, and so on. **read** looks for a linefeed. If there are less words in the input than read variables, it doesn't populate the other variables. If there are more words than variables, the remaining words are placed in the last variable.
read -p "*prompt message*" *variableList*	This shows the *prompt message* and read the words into the variables represented by *variableList*. Those variables and input work the same as without the **-p** and prompt.

```
# Use vi to create the following file called simpRead
$ vi simpRead
read -p "Enter some words:" line
echo "line = $line"
read -p "Enter your first and last name:" first last
echo "first = $first, last = $last"
read -p "Enter first name only, but read asked for 2:" first2
last2
echo "first2 = $first2, last2 = $last2"
read -p "Enter more than 2 words:" one other
echo "one = $one, other = $other"
$
# change the permissions and then execute it
Enter some words:one two three
line = one two three
Enter your first and last name:bob wire
first = bob, last = wire
Enter first name only, but read asked for 2:bob
first2 = bob, last2 =
Enter more than 2 words:bob and barb wire
one = bob, other = and barb wire
```

Reading Text from stdin until EOF (in bash)

read returns a **success** exit status if it reads a line of input.

```
while commam; do
    do something with it
done
```

Note that you can tell the while loop to use a different file for stdin by specifying < and a *fileName* after the **done**.

```
while read line; do
    do something with it
done < fileName
```

```
# copy the file mySentences from
# /usr/local/courses/ssilvestro/cs3423/shell to your directory

#create this script and change its permissions
$ vi rloop
#!/bin/bash
count=0
while read line; do
    echo $line
    let count+=1
done
echo "$count lines"
```

```
$ chmod +x rloop
# Invoke rloop using mySentences for stdi
$ ./rloop <mySentences
Scooby Doo shook with fear when he saw the ghost. Shaggy ran and hid
in the Mystery van. Freddie tried to act brave to impress Daphne, but
she was lovinly watching Scooby Doo. Velma was studying the foot
prints in the mud.

Velma said, "this is the same mud that we saw on the stairs at UTSA!"
6 lines
```

Flow Control Statements in bash
The **while, if, case,** and **for** statements are used for flow control. The various shell dialects have different syntax for flow control.

```
    while condCommand; do
        doSomething
    done

    if condCommand; then
        doSomething
    fi

    if condCommand; then
        doSomething
    else
        doSomethingWhenNotTrue
    fi


    if condCommand; then
        doSomething
    elif condCommand2; then
        doSomething2
    …
    else
        doSomethingN
    fi
```

```
# use vi to create this simple script which sums integers from
# 1 to n
$ vi simpWhile
#! /bin/bash
# check for too few arguments, be careful of the spacing
if [ $# -lt 1 ]; then
    echo "usage: simpWhile number"
    echo "       where number is a number"
    exit 1
fi
sum=0
index=$1
# the spacing is important
while [ $index -gt 0 ]; do
    let sum=sum+index
    let index=index-1
done
echo $sum
$ ./simpWhile 7
$ chmod +x simpWhile
$ ./simpWhile 4
10
$
```

Conditional Commands
while and **if** statements use conditions, which can be commands. There is a special *command*, **test,** which can be used to test conditions:

```
    if test $var -gt 100; then
```

Test Conditional Operators

op1 -gt *op2* *op1* -lt *op2* *op1* -eq *op2*	Numeric comparisons of the two operands. With numeric comparisons, 12 > 2.

The test command returns an exit status of 0 to represent that the condition is true if the variable is greater than 100.

Some people prefer using single brackets around the condition:

```
    if [ $var -gt 100 ]; then
```

Both **test** and the surrounding single brackets cause bash to invoke the **test** command.

bash-only scripts can use double brackets which is handled by bash instead of invoking the test command. Problems associated with using ">" or "<" for test are avoided when using the double brackets.

```
    if [[ $var -gt 100 ]]; then
```

op1 -ne *op2* *op1* -le *op2* *op1* -ge *op2*	
op1 > *op2* *op1* < *op2* *op1* == *op2* *op1* != *op2*	String comparisons of the two operands. With string comparisons, 12 < 2 since the character "1" < "2". See the warning below.
-d *filename*	exists and is a directory
-e *filename*	exists
-f *filename*	exists and is a file not a directory
-r *filename*	exists and is readable
-s *filename*	exists and has a size > 0 bytes
-w *filename*	exists and is writable
-x *filename*	exists and is executable

Warning!
The following example does not do what is expected:

```
    if [ $varA > $varB ]; then
    if test $varA > $varB; then
```

With test and the single bracket expressions, the ">" is interpreted as **redirection** of output to the file named as the value of $varB.

The problem on the left can be avoided with one of these approaches:

- `if [ $varA \> $varB ]; then`
- `if [[ $varA > $varB ]]; then`

case statement
The **case** statement is very powerful. It has multiple patterns for matching values. Syntax:

```
    case "variableRef" in
        pattern1)
            doSomething1
            ;;
        pattern2)
            doSomething2
            ;;
        …
        *)
            doSomethingDefault
            ;;
    esac
```

Each pattern can include

```
#create this exCase file  (just type some of the months)
$ vi exCase
#!/bin/bash
read -p "Enter Month (MMM) DayOfMonth and Year:" month day year
# convert the alpha month to a numeric
case "$month" in
    [Jj]an) mon=1;;
    [Ff]eb) mon=2;;
    [Mm]ar) mon=3;;
    [Aa]pr) mon=4;;
    [Mm]ay) mon=5;;
    [Jj]un) mon=6;;
    [Jj]ul) mon=7;;
    [Aa]ug) mon=8;;
    [Ss]ep) mon=9;;
    [Oo]ct) mon=10;;
    [Nn]ov) mon=11;;
    [Dd]ec) mon=12;;
```

simpleValue	This is just a string to match (e.g., Jan, Feb)
*	This matches anything. By itself, this is used for the default case.
alt1|*alt2*	This specifies alternatives (e.g., dog|cat)
[*list*]	This specifies a list of possible values (e.g. [Jj]an, [Ff]eb). You can also use hyphen for a range of values (e.g., [A-Z]).
?	This matches any single character.

```
    *)      echo "Bad month value = '$month'"
            exit 1;;
esac
echo "date=$mon/$day/$year"
$ chmod +x simpCase
$ ./simpCase
```

for statement

The **for** statement iterates over a list. Syntax options:

```
for var in "$@"; do
    doSomething
done

for var in valueList; do
    doSomething
done

for var in $(command args); do
    doSomething
done
```

Sam's note: another, C-style for-loop utilizing a special syntax of the arithmetic evaluation operator:

```
for ((x=0; x<10; x++)); do doSomething; done
```

```
#use vi to create each of these
$ vi simpFor
echo "1. show the list of files"
for file in "$@"; do
    echo "$file"
done

echo "2. show the list of fruit"
for fruit in orange apple grape; do
    echo "$fruit"
done

echo "3. show the list of files from a command"
for file in $(ls -a *); do
    echo "$file"
done
$ chmod +x simpFor
$ ./simpFor simp* | more
```

break and continue

The **break** and **continue** statements can be used within **for, while,** and **until** statements.

break exits the loop. **continue** continues with the next iteration skipping the remaining statements within the current iteration.

```
#!/bin/bash
# loop until one of the arguments is not a valid file
for file in "$@"; do
    if [ ! -r "$file" ]; then
        break
    fi
    cat $file
done
```

prompted input loop until EOF with multiple prompts

Suppose you want to prompt a user for input or terminate with CTRL-D in a loop. If there are multiple prompts (like a menu), you may need to

```
$ vi showMenu.bash
#!/bin/bash
go=0
while [ $go ]; do
```

do a while that itself doesn't have a terminating condition. When the EOF is encountered, break the loop.

```
    echo "Enter your choice or CTRL-D"
    echo "A - I want do get an A"
    echo "B - I want to get a B"
    echo "F - I give up"
    if ! read ans; then
         # got EOF
         break
    fi
    case "$ans" in
         A) echo "you got an A"
            break
            ;;
         B) echo "you can do better than a B"
            ;;
         F) echo "keep trying"
            ;;
         *)  echo "we think you can type better than that"
    esac
done
```

Reading several variables from multiple lines in a file
Sometimes it is necessary to read several variables from multiple lines of text in a single file. We saw earlier how we can read multiple lines of text from a file using a while loop. This is different since we need to do multiple reads, but have each populate different variables.

Copy the data file that the example uses to your directory:
/usr/local/courses/ssilvestro/cs3423/shell/111

```
$ vi first.bash
#!/bin/bash
read -p "Enter the filename:" filename
# we want to read multiple lines from that file, populating multiple variables
# per line
if [ ! -r $filename ]; then
    echo "could not read that file"
    exit 1
fi
bash second.bash < $filename
```

```
$ vi second.bash
#!/bin/bash
# this reads from stdin
# the file contains:
#   first line:  studentId studentMajor
#   second line: studentName
#   many lines: courseNr courseGrade
read studentId studentMajor
read studentName
echo "Student: $studentId $studentName"
echo "Major: $studentMajor"
echo "Courses:"
# read and echo courseNr and coursegrade until EOF
??

$ bash first.bash
Enter the filename:111
Student: 111 Sally Mander
Major: BIO
Courses:
BIO3233 A
BIO3343 B
BIO1111 A
MAT1214 C
```

Functions

As with many languages, most shells offer the ability to create functions. In Bash, the following syntax creates a function which can be used later

```
my_function () {
    # function body
    echo "hello world"
}
```

To invoke the function, use the name *without* the parentheses

If you want the function body to execute in its own subshell (i.e., not have it affect the current shell's environment), replace the curly brackets with parentheses.

```
my_function2 () (
    # function body
    echo "hello world"
)
```

As with any command, you may use redirection with function invocations.

```
$ vi functions.bash
#!/bin/bash
cd_courses () {
    cd ~/courses
    echo "I'm executing this from within `pwd`"
}

cd_courses
echo "Now I'm executing this from within `pwd`"
$ cd ~
$ bash functions.bash
I'm executing this from within /home/ssilvestro/courses
Now I'm executing this from within /home/ssilvestro/courses

$ vi functionsSubshell.bash
#!/bin/bash
cd_courses () (
    cd ~/courses
    echo "I'm executing this from within `pwd`"
)

cd_courses
echo "Now I'm executing this from within `pwd`"
$ cd ~
$ bash functionsSubshell.bash
I'm executing this from within /home/ssilvestro/courses
Now I'm executing this from within /home/ssilvestro
```