Lecture 2: Shell Programming

Dr. Naureen Nizam
nnizam@cs.toronto.edu

CSCB09H3: Software Tools and Systems Programming (SY 110)
Department of Computer and Mathematical Science
Jan 14/16, 2019

Tomorrow is created here.

UNIVERSITY OF TORONTO SCARBOROUGH
1265 Military Trail, Toronto, Ontario M1C 1A4

Administrivia

- Tutorials/Labs
  - Tutorial 3002 and 3006 are cancelled
  - All Tutorials/Labs will start this week (Tuesday, Wednesday and Thursday)
  - TA information on Quercus along with their email address
  - Lab 1 exercises posted
- A1
  - Will be posted this week.
  - Due: Jan 27, 2019

Shells are very powerful

- Execute all the commands we have already seen (ls, cd, pwd, rm, mv, man, chmod, wc,..)
- Automate things (shell scripting)
- A few other very useful features:
  - I/O redirection
  - Pipelining of commands
  - Filtering output of commands
  - Job control

Input and output redirection

- By default, programs read from standard input (keyboard), write results on standard output (screen), and write errors to standard error (screen)
- You can redirect input, output and errors:
  - " > filename" redirects output to file
  - ">> filename" appends output to file
  - "< inputfile" redirects input (reading from inputfile instead of keyboard)
- 1 before > means stdout (standard output), 2 means stderr (error output)

- `ls >output.txt` # saves the output of ls in output.txt
- `ls -z >output.txt` # does not save output in output.txt …
- `ls -z 2>output.txt` # saves output in output.txt as it redirects stderr

Pipelines

- Use "|" to send the output of one command to the input of another command

- E.g. to count the number of lines produced by "`ls -l`":
  `ls -l | wc -l`
- E.g. to output in sorted order the first 10 lines of a file:
  `head -n 10 file.txt | sort`

Filters

- A filter reads from standard input, processes the input and writes on standard output
- Some useful filters (read about them using `man`)
  - `wc:` count words, lines, characters
  - `grep:` filter lines that do or do not match a pattern
  - `uniq:` remove repeated lines
  - `sort:` sorts input
  - `head:` output only the first lines of the provided input
  - `tail:` output only the last lines of the provided input
  - `sed:` a stream editor to perform text transformations

Job control

- A job (or process) is program in execution
  - Use `ps` to view processes
- Foreground job: has control of the terminal
- Background job: runs concurrently with shell in the background
  - To run a program in the background append & to the name of the program
- At any point a program can be running or suspended
  - Hit `<ctrl> z` to suspend the current foreground job

Job control

- `jobs` gives you a list of jobs; each is associated with a job number
- `fg [num]` puts job `num` in the foreground
- `bg [num]` puts job `num` in the background
- `kill %num` kills job `num`
  - You can kill the current foreground job with Ctrl-c
  - You can suspend (pause) current foreground job with Ctrl-z

Today:

- Shell programming!

- Why? => Automate things

File expansion

- `*` means zero or more characters
- `?` means “exactly one character”
- `[x-y]` means “one character in the range x to y”
- `[^oa]` means “any char except o or a
- `~` means home directory
- `~u` means home directory of user u

```
rm *
ls *.txt
cp ??  ~
cp ~nizamnau/*[0-9] .
```

A first shell program

my_shellscript.sh

```
#!/bin/bash
cd    # a comment
pwd   # another comment
ls
```

- Scripts start with #!/bin/bash
  - This is the path to your bash interpreter
  - Not the same on every machine (use the `which` command to find path)
  - Tells the shell how to interpret/execute the commands in this file
- Followed by shell commands
  - You can use any of the commands we have seen so far
  - Anything you would type in interactive mode into your shell can be put into your shell program
- Mark program file as executable (remember `chmod`?)
- Run from the commandline:

```
nizamnau@mathlab:~$ ./my_shellscript.sh
```

Shell programming

- Shell scripts are useful for automating a series of commands
- But somehow this does not really feel like programming yet…
- What is missing?
  - Variables
  - Input/output (e.g. printing to the screen, reading from the screen)
  - Command-line arguments
  - Control flow
    - If statements
    - For loops
    - While loops
  - Functions

Variables

- Assignment: var=value
  - Important: no whitespace before/after the "="
- Get value: $var
- Variables are not declared, just assign a value
- Variables have no type, they can hold any type of data
- Watch out: Accessing a variable that has no value is not an error, you get the emptry string
- (`echo` is the command for printing to the screen)

hello_world.sh

```
#!/bin/bash

foo=589
bar="Hello World"
baz=$bar

echo $baz
echo $qux
```

```
$ ./hello_world.sh
Hello World

$
```

Some built-in variables

- Use `printenv` and you will see a list of built-in variables your shell maintains
- One that is very useful is `PATH`

```
echo $PATH
```

- Why did we have to type `./hello_world.sh` to execute our own shell script, but other executables don't require a full path (e.g. we can use `ls` instead of `/bin/ls`)?
- You can append "." to your `PATH` variable:

```
export PATH=$PATH:.
export PATH=$PATH:/path/to/dir
```

- This will work only for your current session
  - Add it to your ~/.profile file which is executed by your shell when started
  - .profile is one of those "hidden" files that ls only shows with the –a option

Scope of variables

hello_world.sh

```
#!/bin/bash

foo=589
bar=Hey!
baz="Hello World"

echo $baz
echo $qux
```

```
$ qux="Shells are awesome"
$ ./hello_world.sh
Hello World

$ echo $baz

$
```

- Variables defined in a script are lost when the script ends
  - Unless you use `source` to run the script
- The subshell does not have access to the variables of the parent shell, unless you `export` the variable

```
$ export qux="Shells are awesome"
$ source hello_world.sh
Hello World
Shells are awesome
$ echo $baz
Hello World
```

Shell magic with quotes

- What if you want to store the output of a command in a variable?
- Backquotes cause command substitution
  - `` foo=`ls` `` will store the output of running ls in the variable foo
- What if you want to assign a variable a string value containing the characters $ or \`or * or ~
  - ``foo=$bar`` `` will not assign the string ``$bar`` `` to the variable foo. Why?
  - `foo=*` will not assign the character `*` to the variable foo. Why?
  - Instead use single quotes to force shell to take string literally

```
foo='$bar``~*'
echo $foo
$bar``~*
```

- What about double quotes?
  - Expand variables and do command substitution, but nothing else
  - So what is the output of ``echo "$bar`ls`~*"``

Quoting example

```
$ date
$ Thu Sep 19 12:28:55 EST 2012
$ echo Today is `date`
Today is Thu Sep 19 12:28:55 EST 2012
$ echo "Today is `date`"
Today is Thu Sep 19 12:28:55 EST 2012
$ echo 'Today is `date`'
Today is `date`
```

Another Quoting Example

- What do the following statements produce if the current directory contains the following non-executable files?

```
a b c
```

```
$ echo *          a b c
$ echo ls *       ls a b c
$ echo `ls *`     a b c
$ echo "ls *"     ls *
$ echo 'ls *'     ls *
$ echo `*`        Bash:./: Permission denie
```

More on Quoting

- Command substitution causes another process to be created.
- Which is better? What is the difference?

```
src=`ls *`
```

Or

```
src=*
```

Shell programming

- Shell scripts are useful for automating a series of commands
- But somehow this does not really feel like programming yet…
- What is missing?
  - ~~Variables~~
  - Input/~~output~~ (e.g. printing to the screen, reading from the screen)
    - ~~echo~~
    - read
  - Command-line arguments
  - Control flow
    - If statements
    - For loops
    - While loops
  - Functions

read

- read one line from standard input and assigns successive words to the specified variables. Leftover words are assigned to the last variable.

File Name: name.sh

```
#!/bin/sh
echo "Enter your name:"
read fName lName
echo  "First: $fName"
echo  "Last: $lName"
```

```
$ ./name.sh
Enter your name:
Alexander Graham Bell
First: Alexander
Last: Graham Bell
```

How would you read the names from a file instead of standard input?

Shell programming

- Shell scripts are useful for automating a series of commands
- But somehow this does not really feel like programming yet…
- What is missing?
  - ~~Variables~~
  - ~~Input/output~~ (e.g. printing to the screen, reading from the screen)
    - ~~echo~~
    - ~~read~~
  - Command-line arguments
  - Control flow
    - If statements
    - For loops
    - While loops
  - Functions

Commandline arguments

- Commandline arguments are placed in *positional paramaters.*
- $1, $2, … are the first, second, … commandline arguments
  - After $9, use ${10}
- $0 is the name of the script
- $# is the number of commandline arguments
- $* and $@ list all commandline arguments

foods.sh

```
#!/bin/sh
echo arg1: $1
echo arg2: $2
echo name: $0
echo all: $*
```

➡

```
$ foods.sh pizza pasta lasagna
arg1: pizza
arg2: pasta
name: foods.sh
all: pizza pasta lasagna
```

Positional parameters and `set`

- Did you notice that we have not talked about arrays?
- Bourne shell offers only one array: the positional parameters, $1, $2, ….
- The `set` command assigns its parameters to the positional parameters (all previous pos. parameters are thrown away):

```
$ set pizza spaghetti lasagne
$ echo $1 $2 $3
pizza spaghetti lasagne
```

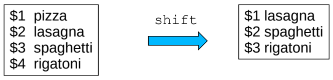
- Useful to store output of commands:

```
$ date
Wed Jan 16 14:37:53 EST 2013
$ set `date`
$ echo The date today is $2 $3, $6
The date today is Jan 16, 2013
```

  - Or use `set `ls`` to store filenames in $1, $2, ….

Positional parameters and `shift`

- Shift moves all positional parameters to the left (so that $1 becomes the old $2, etc.)



$1 pizza
$2 lasagna
$3 spaghetti
$4 rigatoni

shift

$1 lasagna
$2 spaghetti
$3 rigatoni

- What is shift useful for?
  - E.g. iterating over all positional parameters
  - Will see example when we get to for and while loops

Shell programming

- Shell scripts are useful for automating a series of commands
- But somehow this does not really feel like programming yet…
- What is missing?
  - ~~Variables~~
  - ~~Input/output~~ (e.g. printing to the screen, reading from the screen)
    - ~~echo~~
    - ~~read~~
  - ~~Command-line arguments~~
    - Po~~sitional parameter~~s
  - Control flow
    - If statements
    - For loops
    - While loops
  - Functions

The if statement

- In bash the if statement checks the return value of a command and proceeds to “then” if the return value is 0, to “else” otherwise:

```
if   <some command here>
then
    <some commands here>
else
   <some commands here>
fi
```

- What is the return value of a command?
  - It’s NOT the output of the command.
  - Unix requires that programs return 0 for success and some other number for failure.
  - For example, a shell program returns an exit status via the command `exit`
  - You can check exit status of last process with the variable $?

Some examples with if

```
if  ls
then
   echo Exit code $?, job executed fine.
else
   echo Exit code $?, there was a problem.
fi
```

- Normal execution of ls: “`Exit code 0, job executed fine.`”
- No permissions to run ls in this directory: “`Exit code 2, there was a problem.`”

```
if  ls -z
then
   echo Exit code $?, job executed fine.
else
   echo Exit code $?, there was a problem.
fi
```

- -z is invalid option: “`Exit code 2, there was a problem.`”

The `test` command

- `test` takes an expression and returns 0 if its true and 1 if its false.

```
if test $str1 = $str2;
then
   echo "The strings are identical"
else
   echo "The strings are different"
fi
```

- A short form of `test` is [ ]

```
if [ $str1 = $str2 ]
then …
```

- (Note the whitespace after [and before] and before and after = are mandatory)

The `test` command

`-z string`	True if empty string	Tests on strings
`str1 = str2`	True if str1 equals str2	
`str1 != str2`	True if str1 not equal to str2	
`int1 -eq int2`	True if int1 equals int2	Tests on numbers
`-ne, -gt, -lt, -le`		
`-a, -o`	And, or.	
`-d filename`	Exists as a directory	Tests on files /directories
`-f filename`	Exists as a regular file.	
`-r filename`	Exists as a readable file	
`-w filename`	Exists as a writable file.	
`-x filename`	Exists as an executable file.	

expr

- Since shell scripts work by text replacement, we need a special function for arithmetic.

```
x=1
x=`expr $x + 1`
y=`expr 3 * 5`  #doesn’t work
 y=`expr 3 \* 5` #need to escape *
```

- Works only for integer arithmetic
- Can also be used for string manipulation, but we will mostly leave text manipulation for Python.

The while loop

- In bash the while statement executes a command (typically `test`) and keeps looping for as long as the command’s return value is 0.

```
while <some command here>
do
      <some commands here>
done
```

- An example:

```
#!/bin/bash
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
    echo "Please type something in"
    read INPUT_STRING
    echo "You typed: $INPUT_STRING"
done
```

Using while to read from a file

- The following reads one line at a time from the file names “my_file.txt” and prints out each line.

```
#!/bin/bash

file="my_file.txt"

while read line
do
  echo $line
done < $file
```

The for loop

- The for loop loops through every provided value:

```
#!/bin/bash

for i in 1 2 3 4 5
do
    echo $i
done
```

- Values can be anything (not just numbers)
  - … and variable expansion, command substitution, etc. takes place unless you prevent it with the proper quotes

```
#!/bin/bash

for i in $foo `date` ~
do
    echo $i
done
```

A more useful for loop example

- Append to all filenames in the current directory the extension .txt, e.g a file named dummy would be renamed to dummy.txt

```
#!/bin/bash

for i in `ls`
do
    mv $i $i.txt
done
```

- How would you rename files whose names are given to the script as command-line arguments?
  - Need to iterate over positional parameters $1, $2, $3, ….

```
#!/bin/bash

for i in $*
do
    mv $i $i.txt
done
```

Iterating over arguments with while

- Remember the `shift` command?

```
#!/bin/sh
while test "$1″ != “″
do
   echo $1
   shift
done
```

- Don’t use this one unless you know that the argument list will always be short
- sh allows only 9 positional parameters (bash allows ${10}, ${11}, ….)

Shell programming

- Shell scripts are useful for automating a series of commands
- But somehow this does not really feel like programming yet…
- What is missing?
  - ~~Variables~~
  - ~~Input/output~~ (e.g. printing to the screen, reading from the screen)
    - ~~echo~~
    - ~~read~~
  - ~~Command-line arguments~~
  - ~~Control flow~~
    - ~~If statements~~
    - ~~For loops~~
    - ~~While loops~~
  - Functions

Subroutines

- You can create your own functions or subroutines:

```
myfunc() {
  arg1=$1
  arg2=$2
  echo $globalvar $arg1
  return 100
}
globalvar="I like to eat”
myfunc pizza spaghetti
echo $?
echo $arg2
```

```
I like to eat pizza
100
spaghetti
```

- Notes:
  - Arguments are passed through positional parameters.
  - Variables defined outside the function are visible within.
  - Variables defined inside the function are visible outside
  - Return value is stored in $?

A subtlety about return values

- What if we switch the last two commands?

```
myfunc() {
  arg1=$1
  arg2=$2
  echo $globalvar $arg1
  return 100
}
globalvar="I like to eat”
myfunc pizza spaghetti
echo $arg2
echo $?
```

```
I like to eat pizza
spaghetti
0
```

- Notes:
  - Arguments are passed through positional parameters.
  - Variables defined outside the function are visible within.
  - Variables defined inside the function are visible outside
  - Return value is stored in $?

Shell programming

- Shell scripts are useful for automating a series of commands
- But somehow this does not really feel like programming yet…
- What is missing?
  - ~~Variables~~
  - ~~Input/output~~ (e.g. printing to the screen, reading from the screen)
    - ~~echo~~
    - ~~read~~
  - ~~Command-line arguments~~
  - ~~Control flow~~
    - ~~If statements~~
    - ~~For loops~~
    - ~~While loops~~
  - ~~Functions~~

DONE!

That’s it for today!