Software Systems

Lectures Week 3

Bash

Prof. Joseph Vybihal

Computer Science

McGill University

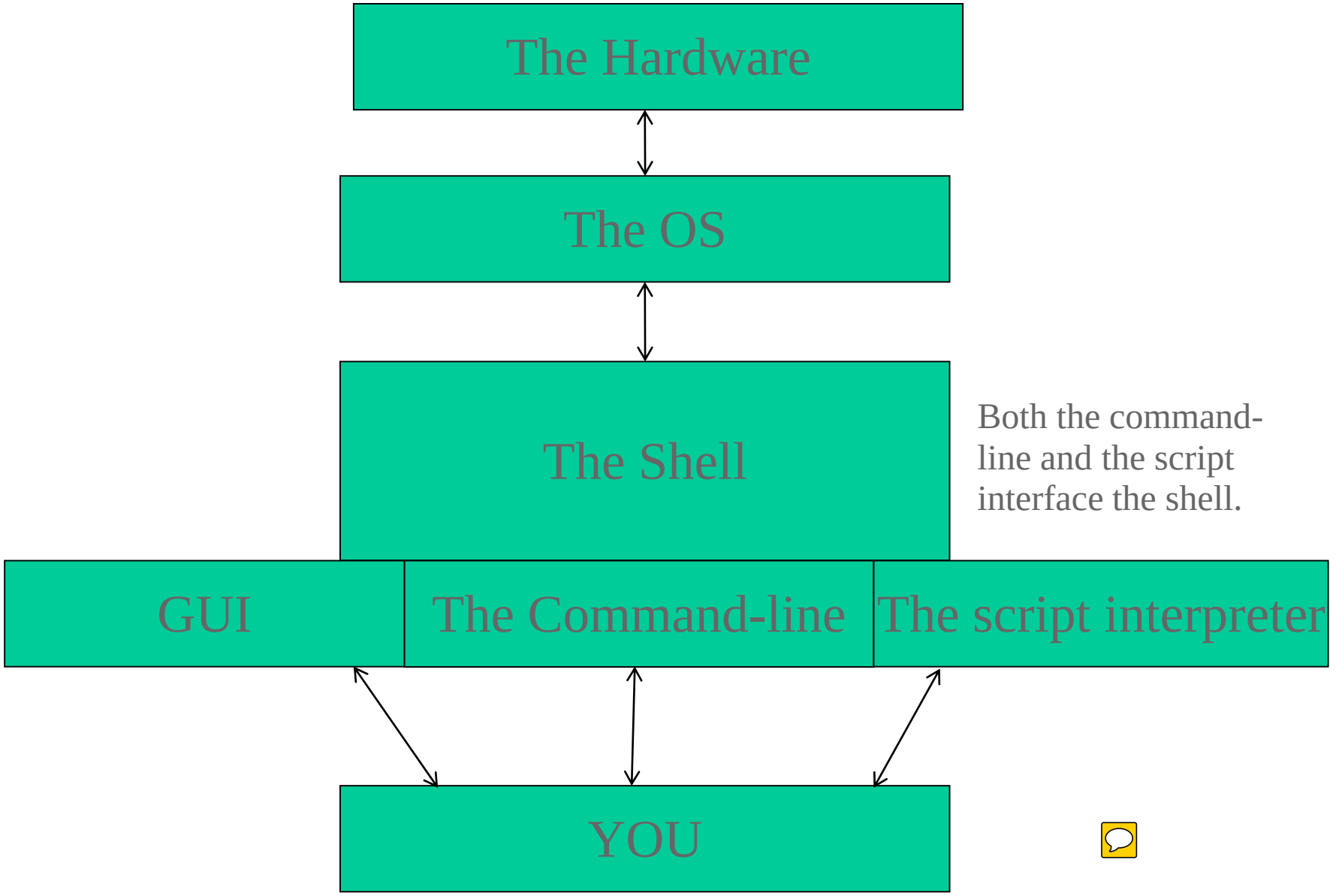
Readings

- Chapter 2 from textbook
- Bash and command-line help:
  - http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
  - http://ss64.com/bash/

Week 3 Lecture 1

Introduction to Bash

The Architecture



Both the command-line and the script interface the shell.

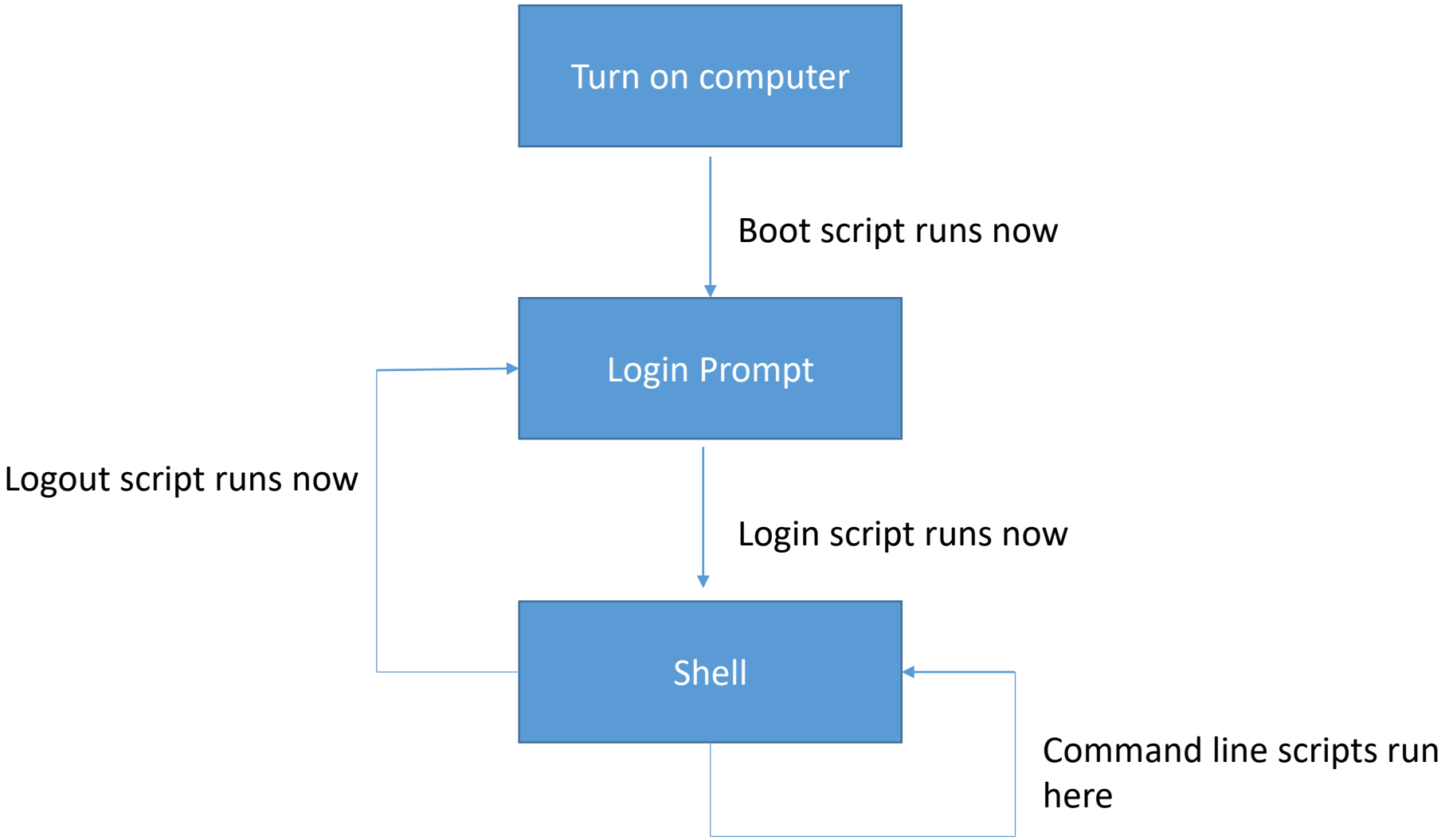
Scripts are used here

- At log in
  - Write scripts to help you get to where you want to go
  - Write scripts to customize the environment
- During development
  - Write scripts that help you to
    - Compile quickly and manage errors and executing the program
    - Copying to and from master
    - Making your own local backups
- At logout
  - Write scripts to do housekeeping
    - Automating backup procedures
    - Automating the logging of events
    - Automating the deletion of files (empty trash)

Two kinds of scripts

- Boot and Login scripts
  - Used to modify the OS environment
  - Boot scripts
    - created by super user for all
  - Login scripts
    - created by owner of account
- Command-line scripts
  - Created by users to automate command-line activities

Two Kinds of Scripts



Scripts

- Scripts are collections of commands, grouped in a file and sequentially execute.
- Scripts are not compiled programs, they are interpreted.
- Scripts run from top to bottom.
- It must be CHMOD'd to execute

Bash

- BASH is a Unix scripting language that implements some programming language control flows, like:
  - functions
  - if
  - for
  - while
- It is interpreted by the OS, not compiled and executed by the CPU.

Example script with demo

```
clear
who
finger bob
ie
outlook
quoteoftheday
```

A script

```
chmod +x file
```

Make text file executable

```
./file
```

Running the script

Remote Issues

clear

who

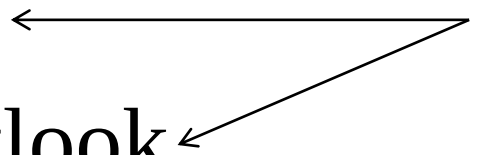
finger bob

ie

outlook

quoteoftheday

```
$ vi bashfile
$ chmod +x bashfile
$ ./bashfile
```



The sha-bang

- The sha-bang #!
  - The first line of the script should start with **#! PROGRAM**
  - Indicates to the OS that the script is to be executed by PROGRAM
    - Different languages can be used to script (sh, bash, perl, python, ruby, etc).
- To set up a Bourne shell script the first line must be:
  - #!/bin/sh
- To set up a Bash shell script the first line must be:
  - #!/bin/bash

Example

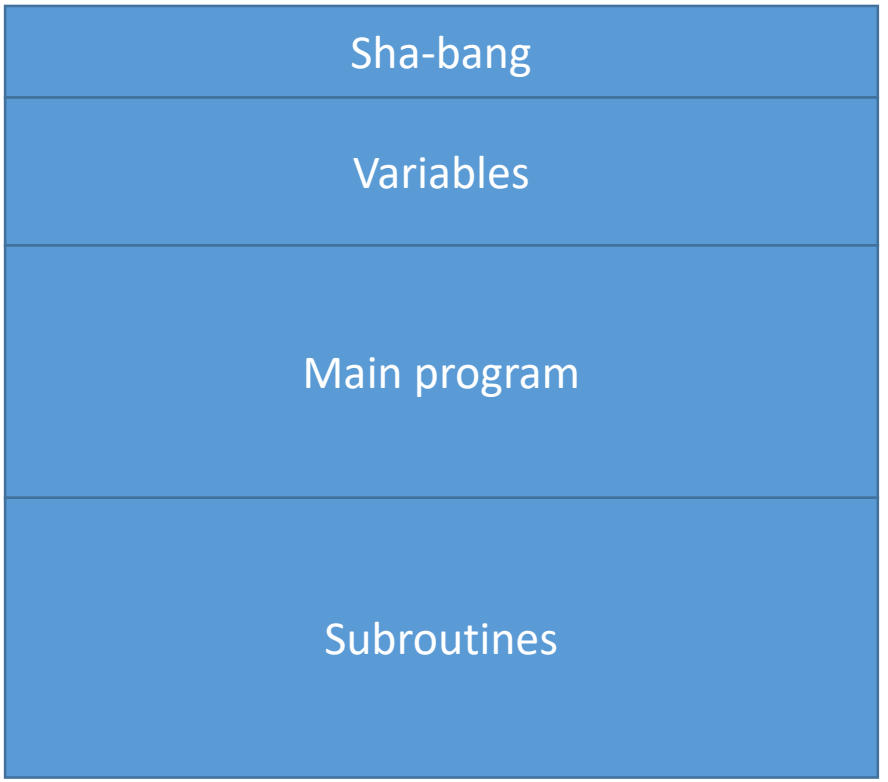
```
#!/bin/bash
clear
who
finger bob
quoteoftheday
```

Declaring that the script interpreter must be Bash

If you do not provide a sha-bang then it will default to the sh shell.

```
$ vi bashfile
$ chmod +x bashfile
$ ./bashfile
```

Bash File Structure



These parts are implicit without strict syntax.

The above organization is customary, however it is not required to be written in this manner. Variables can be declared anywhere and subroutines do not need to be defined at the end of the file.

Bash File Structure

```
#!/bin/bash
clear
who
finger bob
quoteoftheday
```

The sha-bang

The main program

Notice how the script does not have strong syntax requirements. For example the main program does not have a begin-end syntax.

Example

$ vi backup.sh

```
#! /bin/bash

# This is a comment
# Backup files, remove and verify

cp *.txt /home/jack/backup
rm *.txt
ls *.txt
```

$ chmod +x backup.sh
$ ./backup.sh

Example

$ vi morningRoutine.sh

```
#! /bin/bash

# What I like to do each morning

who
chrome http://mail.cs.mcgill.ca
date > today
time >> today
weather >> today
cat today
```

$ chmod +x morningRoutine.sh 
$ ./morningRoutine.sh

Example

$ vi search

```
#! /bin/bash

# Find a file

grep $1 `ls`          # searches within the files for $1
ls | grep $1          # compares the file name for $1
```

$ chmod +x search
$ ./search dog

This becomes $1

Week 3 Lecture 2

Bash Expressions

Variables

- There are four kinds of variables in a shell scripts:
  - Environment Variable : these variables are used to customize the operating system and the shell (use SETENV).
  - User-created : these are script variables (use SET or simply declare the variable: X = 5).
  - Positional Parameters : command line parameters (using $1).
  - Session Variables: defined by the OS or Web Server when the user logs in.

This lecture

- We focus on:
  - User created variables, and
  - Positional parameters
- In lecture 3 week 4:
  - Session, and
  - Environment variables

Positional Parameters

- Passing parameters from the command-line prompt to the script.
  - Eg (no arguments): ./script
  - Eg (arguments) : ./script 5 2 Bob

- From within the script you can access the 5, 2, Bob using the $position notation.
  - X = $1 puts 5 into variable X
  - Y = $2 puts 2 into variable Y
  - Z = $3 puts Bob into variable Z

Prompt: ./bashfilename 5 10 bob

$0 = ./bashfilename

$1 = 5

$2 = 10

$3 = bob

Example

The command ls with positional parameters.

Create the script:
>> vi myls

#!/bin/bash
ls -l-a $1

Just the sha-bang and the ls command, but using the positional parameter

Run the script:
>> ./myls *.doc

What happens internally:
ls -l-a *.doc

Changing *.doc at the command line with *.txt will then list all the txt files.

Positional Variables

- System parameters:
  - $# : number of arguments on the command line
  - $- : options supplied to the shell
  - $? : exit value of the last command executed
  - $$: process number of the current process
  - $! : process number of the last command done in background
- Command-line parameters:
  - $n : where n is from 1 through 9, reading left to right
  - $0 : the name of the current shell or program
  - $* : all arguments on the command line (”$1 $2 ... $9”) as a string
  - $@ : all arguments on the command line (”$1” ”$2” . . . ”$9”) as an array

Script using all positional parameters

•The following script will print out the positional variables:

```
#!/bin/sh
echo "$#:" $#
echo '$-:' $-
echo '$?:' $?
echo '$$:' $$
echo '$!:' $!
echo '$3:' $3
echo '$0:' $0
echo '$*:' $*
echo '$@:' $@
```

What is displayed if this is executed at the prompt?

>> ./pos 3 5 7 8 9

Simple Script Example

•The following script gathers information about the system and stores it in a file specified at the command line.

```
#!/bin/sh
# Assume program name is info.sh
uname -a > $1
date >> $1
who >> $1
```

•The output was as follows :

```
[jvybihal][~/cs206] ./info.sh output.txt

[jvybihal][~/cs206] cat output.txt
Linux rogue 2.6.12-gentoo-r4 #1 SMP ...
Thu Aug 10 10:57:38 EDT 2006
jvybihal pts/0 Aug 10 08:04 (dz2.cs.mcgill.ca)
```

User-created Variables

Declaring and using regular program variables

Declaring in Java is type specific:

```
int x;
x = 10;
```

Script variables work like Java variables except they are un typed.

Declaring in Scripts is type free:

```
x=10
x=”bob”
```

Scripts are sensitive to white spaces.

User-created Variables

Using variables:

```
x="Bob"
echo $x
ls $x
y=$x
```

Note:

echo $y	echo y	echo "y"
Print contents of Y.	Undefined/Unclear.	Print the letter y.

$ vi test.sh

```
# test.sh
x=10
y = 20
echo $x
echo $y
```

Left spaces before and after the equals

$ chmod +x test.sh
$ ./test.sh

```
./test.sh: 2: ./test.sh: y: not found
10
```

Output from running the script.
Notice the error message for y.

Bash Variables and Expressions

- Bash is untyped
  - This means that variables can be assigned to anything
  - This means that variables do not need to be declared
- Bash variables and strings look similar
  - Therefore the need for the $ operator, example:
    - X = 5; X = “Bob”; X = Bob ← the last Bob could be a string or var
    - Solution: X = “Bob”; X = $Bob ← first is a string the second is a var
- Bash expressions have type information
  - Bash uses the square brackets [] and the round brackets () to distinguish type information
  - This is important because we can write command line commands and they can appear as strings or expressions

Mixed Examples

```
X=5             # assign the integer 5
X="Bob"         # assign the string Bob
X=`ls`          # assign to X a list of file names
set `date`      # $0=day, $1=month, $2=year,…
grep $1 `ls`    # in all the files see if $1 exists
```

Capturing Complex Output

•Some commands, such as date, have output that require an extra bit of parsing to use.

```
Sun Aug 13 11:42:38 EDT 2006
```

•You can use the set command to capture and parse the output.

```
set `date`
```

•The output will be stored in $n ($1, $2, $3, etc).

•Note that using set will erase any data you might already have in $n.

Example of set

- The following script executes the date command and outputs the parsed result.

```
#!/usr/bin/sh
set `date`
echo "Time: $4 $5"
echo "Day: $1"
echo "Date: $3 $2 $6"
```

- The output would be as follows:

```
Time: 12:45:54 EDT
Day: Sun
Date: 13 Aug 2006
```


```
set `ls -l stuff.doc`

--rxw--  group   byte
$0       $1      $2
```

Expressions

- Command-line tools
  - Integer tool (expr, with +, -, *, /, =, !=, %, >, <, etc.)
    - expr 5 + 2
    - expr 5 = 5
    - expr $x = $y
    - sum=`expr 5 + 2`
  - BASIC CALCULATOR --- bc
    - If you need to use fractions, use bc to evaluate arithmetic expressions using C programming language syntax.
    - echo $[3/4]
      at the command prompt, it would return 0 because bash only uses integers when answering.
    - Use bc:
      echo 3/4 | bc
      X= `echo 3/4 | bc`
- Bash language syntax

Expressions

- Command-line tools
- Bash language syntax
  - On the command line (or a shell) try this:
    - echo 1 + 1
  - If you expected to see '2' you'll be disappointed.
  - What if you want BASH to evaluate some numbers you have? The solution is this:
    - echo $((1+1))
  - This is to evaluate an arithmetic expression. You can achieve the same effect also like this:
    - echo $[1+1]
    - echo $[“$x”+”$y”]
    - echo $[[$x+$y]] # improved version of the square bracket

Expressions

- Command-line tools
- Bash language syntax
  - $(command) is the same as `command`
    - echo `ls | grep “bob”`
    - echo $(ls | grep “bob”)
  - $((expression))
    - Will evaluate the expression instead of a command (see last slide)
  - ${} expression tool
    - animal=cat
    - echo $animals # No such variable as “animals”.
    - echo ${animal}s
      - cats
    - echo $animal_food # No such variable as “animal_food”.
    - echo ${animal}_food
      - cat_food

Operators

- The ones you are used to:
  - = != + - * / %
- New ones:
  - =~ for regular expressions: X=[[$ANSWER =~ ^y(es)?$]]
  - *,? basic pattern matching: X=[[$ANSWER = y*]]

Quotes

None ---- up to the version of the shell

' ----- as is

“ ---- pre-process first

` ---- exec command, turn ans into string

“ “ “ ---- we will talk about later...

In Bash the ' and the “ are interpreted as the same.

Double Quote

- Preprocess interpretation
  - X='abc\n'
  - Y="abc\n"
  - echo X
    - Outputs: abc\n
  - echo Y
    - Outputs: abc
      <new line>
  - Escape characters:
    - \n new line
    - \t tab
    - \b back space
    - \a alert
    - \\ display a back slash

In Bash you must use:

echo –e "abc\n"

To see the effect.

Writing to STDOUT

The `echo` command writes to STDOUT

echo “text with default carriage return”

echo –e “text with back slash pre processing”

echo –n “text without default carriage return”

What does this print”

echo “hello” echo –n “hello” echo –e “hello\n”

Reading from STDIN

- The `read` command allows you to read a string from STDIN.
- That string is then stored in the specified variable.

```
#!/bin/bash

echo "What is your name?"
read name
echo "Your name is $name."        bob
echo 'Your name is $name.'        $name
```

Question

- Write a script that will tell you if your friend is currently logged into the system.

- Write a script that calculates the number of days you have been alive. Assume 365 days for every year. Do not consider leap years. Ask the user for their age.


```
./days
Years:
20
```

```
#! bash
echo “Years:”
read age
x=`expr $age * 365`
echo $x

echo `expr $age * 365`

echo $(( $age * 365 ))
```

```
vi abc.txt
./days < abc.txt
```

Week 3 Lecture 3

Bash Control Structures

Conditionals in Bash

The test Command

- The test command is used to evaluate a condition.
- The test command can evaluate condition at the file, string or integer level.

File Tests

Example: if (-r .cshrc)

- -r file: true if it exists and is readable
- -w file: true if it exists and is writeable
- -x file: true if it exists and is executable
- -f file: true if it exists and is a regular file
- -d name: true if it exists and is a directory
- -h or -L file: true if exists & a symbolic link
- and many more . . .

String Tests

Example: if ($x)

- -z string : true if the string length is zero
- -n string : true if the string length is non-zero
- string1 = string2 : true if strings are identical
- string1 != string2 : true if strings not identical
- string : true if string is not NULL

Integer Tests

- n1 -eq n2 : true if integers n1 and n2 are equal
- n1 -ne n2 : true if integers n1 and n2 are not equal
- n1 -gt n2 : true if integer n1 is greater than integer n2
- n1 -ge n2 : true if int n1 is greater than or equal to int n2
- n1 -lt n2 : true if integer n1 is less than integer n2
- n1 -le n2 : true if int n1 is less than or equal to int n2

Expressions

- Command-line tools
- Bash language syntax
  - $(command) is the same as `command`
    - echo `ls | grep “bob”`
    - echo $(ls | grep “bob”)
  - $((expression))
    - Will evaluate the expression instead of a command (see last slide)
  - ${} expression tool
    - animal=cat
    - echo $animals # No such variable as “animals”.
    - echo ${animal}s
      - cats
    - echo $animal_food # No such variable as “animal_food”.
    - echo ${animal}_food
      - cat_food

Control Structures

The if Statement

- Bash if-statement behaves similar to Java

```
if _condition_
then
    _code_
elif _condition_
then
    _code_
else
    _code_
fi
```

Example

- The following example program can be used to add or subtract two numbers.

```
#!/bin/sh
if test $1 = “add”
then
   result=‘expr $2 + $3‘
elif test $1 = “sub”
then
   result=‘expr $2 - $3‘
else
   result=0
fi
echo "The result is $result \n"
```

```
% ./calc add 5 10
```

Example

```
#!/bin/bash
# Count=99
if [ $1 -eq 100 ]
then
  echo "Count is 100"
elif [ $1 -gt 100 ]
then
  echo "Count is greater than 100"
else
  echo "Count is less than 100"
fi
```

```
% ./bla 50
```

The case Statement

- A case statement is similar to a Java switch statement.

```
case condition in

    condition1) action1;;
    condition2) action2;;
    condition3 | condition4) action3;;

    *) else_action;;

esac
```

$ ./prog add 5 2
$1 $2 $3

Example

- The following example program is a reformatting of the if example, but with a case statement.

```
#!/bin/bash
case $1 in
“add” | “addition”) result=‘expr $2 + $3‘;;
“sub” | “subtraction”) result=‘expr $2 - $3‘;;
*) result=0;;
esac
echo "The result is $result \n"
```

Bash Example

```
$ cat yorno.sh
#!/bin/bash

echo -n "Do you agree with this? [yes or no]: "
read yno
case $yno in

        [yY] | [yY][Ee][Ss] )
                echo "Agreed"
                ;;

        [nN] | [n|N][O|o] )
                echo "Not agreed, you can't proceed the installation";
                exit 1
                ;;
        *) echo "Invalid input"
            ;;
esac

$ ./yorno.sh
Do you agree with this? [yes or no]: YES
Agreed
```

The for Loop

- The for loop is similar to a Java *iterator*.
- It allows you to iterate (loop) over a list of strings.

```
for var in list
do
    actions
done
```

Example

- The following script executes the file command for each file in the specified path.

```
#!/usr/bin/sh      % bla *.*
for i in `ls $1`
do
    file $i;
done
```


```
$ ls *.doc
F1.doc   f2.doc f3.doc
$ ./bla *.doc
```

```
for i in “f1.doc f2.doc f3.doc”
do
    Do some stuff
done
```

1. do some stuff with i = “f1.doc”
2. do some stuff with i = “f2.doc”

Example

```
$ cat for2.sh
i=1
weekdays="Mon Tue Wed Thu Fri"
for day in "$weekdays"
do
 echo "Weekday $((i++)) : $day"
done

$ ./for2.sh
Weekday 1 : Mon
Weekday 2 : Tue
Weekday 3 : Wed
Weekday 4 : Thu
Weekday 5 : Fri
```

Example

```
$ cat for5.sh
i=1
for file in /etc/[abcd]*.conf
do
 echo "File $((i++)) : $file"
done

$ ./for5.sh
File 1 : /etc/asound.conf
File 2 : /etc/autofs_ldap_auth.conf
File 3 : /etc/cas.conf
File 4 : /etc/cgconfig.conf
File 5 : /etc/cgrules.conf
File 6 : /etc/dracut.conf
```

The while Statement

- Very similar to Java

```
while condition
do
    actions
    [continue]
    [break]
done
```

Using parameters

- The following script will pad a file with zeros.

```
#!/usr/bin/bash

i=`wc -c < $1`;
while test $i -lt $2
do
    echo -n "0" >> $1;
    i=`wc -c < $1`;
done
```

```
% ./fill ass1.c 100
```

Wrap

```
$ more dext
#!/bin/sh

if [ $# = 0 ]
    then
        echo "Usage: $0 name"
        exit 1
fi

user_input="$1"
grep -i "$user_input" << DIRECTORY
    John Doe     555.232.0000    johnd@somedomain.com
    Jenny Great 444.6565.1111   jg@new.somecollege.edu
    David Nice   999.111.3333    david_nice@xyz.org
    Don Carr     555.111.3333    dcarr@old.hoggie.edu
    Masood Shah 666.010.9820    shah@Garments.com.pk
    Jim Davis    777.000.9999    davis@great.advisor.edu
    Art Pohm     333.000.8888    art.pohm@great.professor.edu
    David Carr   777.999.2222    dcarr@net.net.gov
DIRECTORY

exit 0
$ dext
```

$ ./dext Jenny
Jenny Great 444.6565.1111 jg@
$

grep "$1" < filename.txt

Grep $x "john doe jenny great....."