CSCI 2132: Software Development

Shell Scripting

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

Reading

Glass and Ables, Chapter 8: bash

Your Shell vs Your File Manager

File manager

- Easy and intuitive to use (point and click)
- Almost no need to understand how computers work

Shell

- Need to remember commands to achieve certain things
- Typing commands is more efficient than point and click
- Use utilities and pipelines to achieve complicated tasks beyond selecting and copying files
- Shell scripts = programs built out of shell and utility commands to automate complex work flows (create your own “custom commands”)

Shell Variables

- Your shell can store chunks of text in variables for later use.
- Some of these variables are special.
  (Do not mess with them unless you know what you are doing.)
- Set a variable: `var= ...`
- Use a variable: `$var`

Example:

```
$ hello=‘Hello, world!’
$ echo $hello
Hello, world!
```

Customizing Program Behaviour via Shell Variables

- The path where your shell finds programs you try to run:

```
$ env | grep PATH
PATH=/users/faculty/nzeh/bin:/local/bin:/bin:/usr/bin: ...
```

- The path to your shell:

```
$ env | grep SHELL
SHELL=/bin/bash
```

- Your user name:

```
$ env | grep USER
USER=nzeh
```

- The type of terminal you use:

```
$ env | grep TERM
TERM=xterm-256color
```

Customizing Program Behaviour via Shell Variables

- Your default editor:

```
$ env | grep EDITOR
EDITOR=vi
```

- Your CSID (only on bluenose):

```
$ env | grep CSID
CSID=nzeh
```

Capturing Output in Variables

`program1 `program2``:

- Run `program2` and pass its `stdout` as a command line argument to `program1`.

Example:

```
$ echo `echo ‘Hello, world!’`
Hello, world!
$ cd `echo $PATH | cut -d: -f3`
# Now I’m in directory /bin
```

Capture `stdout` in a variable:

```
$ hello=`echo ‘Hello, world!’`
$ echo $hello
Hello, world!
```

Repeating Command Sequences

Compile your Java program, run it, and verify the output:

```
$ javac HelloWorld.java
$ java HelloWorld > HelloWorld.out
$ less HelloWorld.out
```

What if I want to do this often during development?

compile-and-test.sh

```
#!/bin/sh
javac HelloWorld.java
java HelloWorld > HelloWorld.out
less HelloWorld.out
```

```
$ chmod 700 compile-and-test.sh
$ ./compile-and-test.sh
```

Shell Scripts

A **shell script** is a text file containing a sequence of shell (built-in commands or utility programs) commands.

Running a shell script:

- `sh <script file name>`
- `chmod u+x <script file name>; ./<script file name>`
- `. <script file name>`
  (may alter the behaviour of the current shell)

Command Line Arguments

Often, we want to pass arguments to a shell script as if it was a regular program.

Arguments:

- `$0` = program (script) name
- `$1`, `$2`, ... = arguments
- `$#` = number of command line arguments, not counting `$0`

Example:

`compile-and-test.sh`

```
#!/bin/sh
javac $1.java
java $1 > $1.out
less $1.out
```

Arithmetic Operations

Arithmetic expressions to be evaluated must be enclosed in double parentheses:

```
(( expression))
```

Arithmetic operators:

- = (assignment), +, -, ++, --, *, /, % (mod), ** (power)

Example:

```
#!/bin/bash
(( sum = $1 + $2 ))
echo the sum of $1 and $2 is $sum
```

Logical Expressions

In if-statements and while-loops (soon), we need to be able to test logical conditions.

Arithmetic conditions: `(( expression ))`

- Comparison operators: `<=`, `>=`, `<`, `>`, `==`, `!=`
- Logical operators: `!` (not), `&&` (and), `||` (or)

String tests: `[ expression ]` (spaces necessary)

- Comparison operators: `==`, `!=`
- Basic tests: `-n` (not empty), `-z` (empty)
- Logical operators `!`, `&&`, `||`

Repeating Things: `for` Loops

Repeat a given sequence of commands for every element in a list:

```
for <var> in <list>; do <cmd>  ... ; done
```

Example: Rename every file `<file>` to `my_<file>`:

```
$ for file in *; do mv $file my-$file; done
```

Example: Strip the suffix of all .hpp (C++ header) files:

```
$ for file in *.hpp; do \
    mv $file `echo $file; sed -e 's/\.hpp$//'`; done
```

Adding Decisions: `if` Statements

Similar to Java but different syntax:

```
if condition1; then
    commands
elif condition2; then
    commands
else
    commands
fi
```

The elif and else parts are optional.

An Example

```
#!/bin/bash

if (( $# != 2 )); then
    echo usage: $0 num1 num2
    exit
fi

(( sum = $1 + $2 ))
echo the sum of $1 and $2 is $sum
```

Java-Style Arithmetic `for` Loops

```
#!/bin/bash

if (( $# != 1 )); then
    echo usage: $0 num1
    exit
fi

for (( i = 1; $i <= $1; i = $i + 1 )) do
    f=tmpfile-$i.txt
    echo “Appending to file $f”
    echo Updated on `date` >> $f
done
```

Multi-way Branching: `case` Statements

Similar to switch statement in Java:

```
case var in
    word{|word}*)
        commands
        ;;
    ...
esac
```

Example of a `case` Statement

```
#!/bin/bash
day=`date | cut -f1 -d” “`

case “$day” in
    Mon|Wed|Fri)
        echo 2132 lectures
        ;;
    Tue|Thu)
        echo No 2132 lectures
        ;;
    Sat|Sun)
        echo Do 2132 homework
        ;;
esac
```

Repeating things: `while` and `until`

Repeat commands while a condition is true:

```
while condition; do
    command
    ...
done
```

Repeat commands until a condition is true:

```
until condition; do
    command
    ...
done
```

The Earlier `for` Loop Redone Using `while`

```
#!/bin/bash

if (( $# != 1 )); then
    echo usage: $0 num1
    exit
fi

i=1
while (( $i <= $1 )); do
    f=tmpfile-$i.txt
    echo “Appending to file $f”
    echo Updated on `date` >> $f
    (( i = $i + 1 ))
done
```

Conditional Expressions for Status of Files

`[ -e file ]` Does `file` exist?

`[ -f file ]` Is `file` a regular file?

`[ -d file ]` Is `file` a directory?

`[ -r file ]` Is `file` readable?

`[ -w file ]` Is `file` writable?

`[ -x file ]` Is `file` executable?

Again, the spaces after `[` and before `]` are required!

Exit Codes

How does the shell check whether a command you tried to run was successful?

Every program returns an exit code that is 0 on success and some non-zero value on error.

This exit code is assigned to the special variable `$?` after the command runs.

```
$ cp a b; echo $?
cp: a: No such file or directory
1
$ touch a; echo $?
0
```

Returning an Exit Code from a Shell Script

```
exit            Exit the script with error code $?

exit num        Exit the script with error code num
```

Example: A Backup Script

Specification:

- Script takes two arguments:
  a source directory and a destination directory
- Each file from the source directory is copied
  to the destination directory.
- Only regular files are copied (not directories).
- Files are copied if they do not already exist in he destination directory.
- Print the name of each file being copied.

Example: A Backup Script

```
#!/bin/bash

if [ ! -d $1 ]; then
    echo Source directory does not exist
    exit 1
elif [ ! -d $2 ]; then
    echo Destination directory does not exist
    exit 1
fi

for filename in `ls $1`; do
    if [ -f $1/$filename ]; then
        if [ ! -e $2/$filename ]; then
            cp $1/$filename $2/$filename
            echo $filename
        fi
    fi
done
```