Shell Scripting

Learning Outcomes

At the end of this session, the students should be able to:

1. Understand the basics of Linux Shell; and
2. Implement a shell script using the Linux shell

Content

Introduction

The **shell** is an interpreter that executes the commands given by the user. The shell accepts the user's input and processes it for the operating system to perform. The **terminal** is a window that allows you to interact with the shell.

There are a lot of shell programs for Linux including **Bourne-Again Shell (bash), Korn Shell (ksh), C Shell (csh)**, and **Z Shell (zsh).**

Shell Scripting

Shell scripting allows users to program commands in chains. In a shell script, you can use the output of a command as an input or even branch out depending on one command's output. Any command used in the terminal is a valid command in a shell script.

For the laboratory, we will be focusing on the **Bourne-Again Shell**.

General Rules

1. Every script should begin with the following line: #!/bin/bash
2. Each new line in a shell script is a new command
3. Comments starts with the pound symbol (#).
4. As a precaution, always "echo" your script first before executing it.

Variables

There two (2) types of variables in shell scripting:

1. **Environment Variables**
   These variables are created by the operating system itself and are defined in CAPITAL LETTERS. The command used to view all environment variables is **printenv**. You can also add your own environment variable by using the command **export**.

2. **User-Defined Variables (UDV)**
   These variables are created by the users and are used only for the script. Also, these are usually defined in lower case letters and start with letters or an underscore followed by alphanumeric characters.

Initializing and accessing shell variables are quite different from the usual programming languages. In order to initialize variables, it should follow the syntax **<var name>=<value>**. Take note that there is no space before or after the equal sign. On the other hand, accessing variables should always be prepended by **$**.

Example

```
$ printenv
$ export MYNAME="this is my name"
$ printenv | grep MYNAME
$ var1="Hello World"
$ echo var1; echo $var1
$ var2=var1; var3=$var1
$ echo $var2 $var3
```

Parameter Expansion

Placing a variable inside a pair of curly braces allows simple string manipulations to it. Below are some of the parameter expansions you can perform. These commands return a new string without changing the original value of the variable. The syntax also protects the variable which allows it to be next to any characters when printing it.

Parameter Expansion Syntax	Result
`${var:m}`	Gets the substring of *var* starting at index *m* (zero-based indexing)
`${var:m:n}`	Gets the substring of *var* starting at index *m* up to *n* characters
`${var/str1/str2}`	Replaces the first occurrence of *str1* with *str2*
`${var%suffix}`	Removes *suffix* from var
`${var%/*}`	Removes the trailing file from path (*path/to/file.txt* will be *path/to*)
`${var##*/}`	Gets the basepath (*path/to/file.txt* will be *file.txt*)

Example

```
$ i=10
$ echo $ith integer
$ echo ${i}th integer
$ file=main.c
$ line='this is my script'
$ echo ${file:2}
$ echo ${line:(-6)}
$ echo ${file:2:2}
$ new_line=${line/is/at}
$ echo ${line}; echo ${new_line}
$ gcc ${file} -o ${file%.c}_program
```

Quotes

There are three (3) kinds of quotes in shell scripting:

1. **Double Quotes**
   Double quotes **allows** the evaluation of the variable that uses the dollar sign ($) or the backlash (\).
2. **Single Quotes**
   Single quotes **does not allow** the evaluation of a $ or a \.
3. **Backticks/Backquotes(`)**
   Backticks are used to **execute** the command that it contains. This is similar with enclosing the command in $(<command>).

Example

```
$ line=World
$ var1="Hello ${line}"
$ var2='the ${line}'
$ echo $var1; echo $var2
$ echo `whatis echo` $(whatis echo)
```

Getting Input

The command **read** gets input from the user via the interactive shell.

Example

```
$ read n
$ echo $n
```

Running the script

A shell script containing the chain of commands is made to be executable via the **chmod** command. If there are changes in the file, there is no need to use the chmod command again if it is already an executable. Running the script using the **./** notation creates a subprocess or another shell. Moreover, this method can be extended by creating a symlink of the executable to any directory found in $PATH so that you can run it directly using the symlink. There are more ways to run a script such as creating a shell function and introducing an alias.

Example files:

```
# contents of my_script.sh

#!/bin/bash

gcc main.c
./a.out
```

Example usage:

```
# make sure the terminal is at 1run directory
$ chmod +x my_script.sh
$ ./my_script.sh
```

Positional Arguments

Using command-line arguments makes the shell script dynamic and flexible depending on the inputs.

Variable	Result
`$1, $2, ..., ${10}, ${N}`	Gets the *N*th argument from the command line
`$0`	Reserved for the command or shell script called
`$@`	Gets the collection of all positional arguments except $0

Example:

```
# make sure the terminal is at 2args directory
$ ./2args.sh lorem ipsum dolor sit amet
$ ./2args.sh lorem ipsum "dolor sit" amet
```

Arrays

The implementation of arrays in shell scripting is different from most programming languages. You should always specify if you are accessing an element or the whole array in its parameter expansion. Surprisingly, the special variable for all positional arguments ($@) is not an array.

Example:

```
$ array=( 1 two 3 four ) # notice the spaces
$ echo ${array}
$ echo ${array[@]}
$ echo ${array[0]} ${array[1]}
$ echo ${#array[@]}A
$ ./3array.sh q w e r t y # make sure the terminal is at 3array directory
```

Conditional Statements

Chaining commands are often seen in shell scripting and its execution is not stopped if an error has occurred. This is dangerous especially if the success of a command (moving or updating files) is necessary for the next commands to be successful as well. Conditional execution of commands helps in these kinds of situations.

Conditional execution syntax:

Operator	Usage
`COMMAND1 && COMMAND2`	*COMMAND2* is only executed if *COMMAND1* is successful (or returned true)
`COMMAND1 || COMMAND2`	*COMMAND2* is only executed if *COMMAND1* is not successful (or returned false)

Example:

```
# make sure the terminal is at 4conditional directory
$ cat does_not_exist && echo this will not be printed
$ cat does_exist || echo this will be printed
```

The whole conditional statement in shell scripting is enclosed by the keywords **if** and **fi**. Each condition must be enclosed by double square brackets ([[condition]]) and should be followed by a **then** keyword. Here is the syntax for the if-else statement in shell scripting:

```
if <conditional_expression> ; then
    # do something
elif <conditional_expression> ; then
    # do something
else
    # do something
fi
```

Here are some of the conditional expressions:

Conditional Expression	Returns true if
`[[ -z STRING ]]`	*STRING* is empty
`[[ STRING1 == STRING2 ]]`	*STRING1* and *STRING2* are equal
`[[ NUM1 -eq NUM2 ]]`	*NUM1* and *NUM2* are equal
`[[ NUM1 -ne NUM2 ]]`	*NUM1* and *NUM2* are not equal
`[[ NUM1 -gt NUM2 ]]`	*NUM1* is greater than *NUM2*
`[[ NUM1 -ge NUM2 ]]`	*NUM1* is greater than or equal to *NUM2*
`[[ -d DIR ]]`	*DIR* is a valid directory
`[[ -f FILE ]]`	*FILE* is a file (not a directory)

You can also use these conditional statements in conditional execution.

Loops

Loop construct follows the syntax below:

```
# a C-like loop
for (( i=0; i<$num; i++)); do
   echo $i
done

# another example
for i in command; do
   echo $i
done
```

The second example is used more in shell scripting since you can use commands, arrays, and even the $@ variable in it.

Learning Experiences

A simple shell script will be implemented.

Assessment Tool

A programming assignment that requires the student to create a shell script.

References

[1] Bash scripting cheatsheet. (n.d). Retrieved August 28, 2020, from https://devhints.io/bash
[2] The Bash Hackers Wiki. (n.d.). Retrieved August 28, 2020, from https://wiki.bash-hackers.org/start