CHAPTER 10, LAB 1: PROGRAMMING THE BOURNE AGAIN SHELL (30 MINUTES)

LEARNING OBJECTIVES AND OUTCOMES

In this lab you will learn to write more advanced shell scripts that work with command-line arguments, control structures, and temporary files. Using read, these scripts will read information the user enters and, using echo, display information based on what the user enters. You will also learn about processes, PIDs, and using export to modify the environment.

READING

Read the sections on parameters (Sobell, pages 460–470), control structures (Sobell, pages 419–435), and environment (Sobell, pages 471–474).

PROCEDURE

The shell expands **$0** to the name of the calling program (Sobell, page 461), **$1–$*n*** to the individual command-line arguments (positional parameters; Sobell, page 462), **$*** (Sobell, page 465) to all positional parameters, and **$#** (Sobell, page 466) to the number of positional parameters.

1. Write a script named **all** that displays (sends to standard output) the name of the calling program, the number of positional parameters, and a list of positional parameters. Include the **#!** line (Sobell, page 287) and a comment (Sobell, page 288). Remember to make the file executable (Sobell, page 100). Test the script with 0, 1, and 5 positional parameters.

```
$ cat all
#!/bin/bash
#
# Program to display its name, the number of arguments it
# was called with, and a list of those arguments.
#
echo "This script was called as $0."
echo "This script was called with $# arguments."
echo "The arguments were: $*"

$ ./all
This script was called as ./all.
This script was called with 0 arguments.
The arguments were:

$ ./all one
This script was called as ./all.
This script was called with 1 arguments.
The arguments were: one
```

```
$ ./all 11 12 13 14 15
This script was called as ./all.
This script was called with 5 arguments.
The arguments were: 11 12 13 14 15
```

2. Make a symbolic link (Sobell, page 113) named **linkto** to the **all** script you wrote in the previous step. Call **linkto** with two arguments. What does the script report as the name it was called as?

```
$ ln -s all linkto
$ ./linkto one two
This script was called as ./linkto.
This script was called with 2 arguments.
The arguments were: one two
```

3. Write a script named **myname** that uses echo (most of the examples in Chapter 10 use this utility) to prompt the user with **Enter your name:** , reads into a variable the string the user types in response to the prompt, and then displays **Hello** followed by the string the user typed. When you run the program it should look like this:

```
$ ./myname
Enter your name: Max Wild
Hello Max Wild
```

```
$ cat myname
#!/bin/bash
#
# Program to prompt the user, read the string the user types,
# and display the string the user typed
#
echo -n "Enter your name: "
read ustring
echo "Hello" $ustring
```

4. Rewrite **myname** from the previous step, calling it **myname2**. Have this version of the program prompt the user for a string, but instead of displaying **Hello** and the string on the screen (sending it to standard output) redirect the output so the program writes only the string the user entered (and not **Hello**) to the temporary file named ***PID*.name** where ***PID*** is the process ID number (Sobell, page 467) of the process running the script. Display the contents of the ***PID*.name** file.

```
$ cat myname2
#!/bin/bash
#
# Program to prompt the user, read the string the user types,
# and write the string the user enters to $$.name
#
echo -n "Enter your name: "
read ustring
echo $ustring > $$.name
```

```
$ ./myname2
Enter your name: Max Wild

$ ls
18218.name  cptobak  first.bak  mine    myname2
all         first    linkto     myname  short
$ cat 18218.name
Max Wild
```

5. Write and run a script named **looper** that uses the **for** (Sobell, page 434) control structure to loop through the command-line arguments and display each argument on a separate line.

```
$ cat looper
#!/bin/bash
#
# Program to display its arguments one per line
#
for idx
do
     echo $idx
done

$ ./looper a b c d
a
b
c
d
```

6. Rewrite **looper** from the previous step, calling it **looper2**. Have this version of the program use the **for...in** control structure (Sobell, page 432) to perform the same task.

```
$ cat looper2
#!/bin/bash
#
# Program to display its arguments on one line
#
for idx in $*
do
     echo $idx
done
```

7. Write a script named **ifthen** that prompts the user with **>>** and reads a string of text from the user. If the user enters a nonnull string, the script displays **You entered:** followed by the string; otherwise it displays **Where is your input?**. Use an **if...then...else** control structure (Sobell, page 424) to implement the two-way branch in the script. Use the test (Sobell, pages 420 and 978) builtin to determine if the user enters a null string. What do you have to do to avoid getting an error message when you prompt with **>>**? (There are several ways to construct the test statement.)

```
$ cat ifthen
#!/bin/bash
#
# Program to prompt the user and display the string the
# user entered. If the user enters nothing, query the user.
#
echo -n ">> "
read userinput
if [ "$userinput" == "" ]
     then
         echo "Where is your input?"
     else
         echo "You entered: " $userinput
fi
```

You must quote the **>>** characters to prevent the shell from interpreting these characters and generating an error message when you display this prompt.

8. Write and run a simple shell script named **echomyvar** that displays the PID (Sobell, page 467) of the process running the script and value of the variable named **myvar**. Display the PID of the interactive shell you are working with. What is the value of the variable within the process running the shell script? Are the interactive shell and the shell running the script run by the same or different processes (do they have the same PID)?

```
$ cat echomyvar
echo The PID of this process is $$
echo The value of myvar is: $myvar
$ echo $$
2651
$ ./echomyvar
The PID of this process is 4392
The value of myvar is:
```

The value of the **myvar** variable within the process running the shell script is null. The interactive shell and the script are run by different processes (that have different PIDs).

The example near the top of Sobell, page 473, demonstrates a way to put a variable in the environment of a script you are calling without declaring the variable in the interactive shell you are running. Use this technique to assign a value to **myvar**, place it in the environment of **echomyvar**, and run **echomyvar**. After running the script, is **myvar** set in the interactive shell?

```
$ myvar=sam ./echomyvar
The PID of this process is 4411
The value of myvar is: sam
$ echo $myvar

$
```

No, **myvar** is not set in the interactive shell.

On the command line, assign a value to the variable named **myvar** and then run **echomyvar** again. Which value does the script display?

```
$ myvar=zach
$ ./echomyvar
The PID of this process is 4428
The value of myvar is:
```

The script displays a null value.

Use the export (Sobell, page 472) builtin in the interactive shell to place **myvar** in the environment and run **echomyvar**. (You can export **myvar** without assigning a new value to it.) Did the script display the value of **myvar** this time? Is the PID of the interactive shell the same or different from the PID of the script?

```
$ export myvar
$ ./echomyvar
The PID of this process is 4452
The value of myvar is: zach
$ echo $$
2651
```

Yes, the script displayed the value of **myvar**; the PIDs are different.

Call export without an argument and send the output through grep to display the export attribute for **myvar**. The export builtin and **declare –x** perform the same function.

```
$ export | grep myvar
declare -x myvar="zach"
```

Use **export –n** (Sobell, page 473) to unexport **myvar**. Show that **myvar** is a shell variable but is no longer in the environment.

```
$ export -n myvar
$ echo $myvar
zach
$ export | grep myvar
```

Use the unset (Sobell, page 304) builtin to remove **myvar**. Show that **myvar** is not available in the interactive shell or in the environment.

```
$ unset myvar
$ echo $myvar
$ export | grep myvar
```

DELIVERABLES

This lab gives you practice using positional parameters, comments, temporary files, and control structures in shell scripts that query the user and read information the user enters. It also has you work with PIDs, the export builtin, and the environment.