Birzeit University - Faculty of Engineering & Technology
Electrical & Computer Engineering Department - ENCS313
Linux laboratory

Experiment #4
Shell scripting
Part I

0.1 Introduction

The experiment intends to present to students the shell scripting. Students will be shown how to take advantage of the Linux commands they have seen so far in order to build shell scripts. Students will be shown as well how to create and use variables, will be exposed to built-in integer arithmetic, the use of single and double quotes as well as the usage of the backslash symbol. Command substitution and special shell variables will be discussed. Afterwards, the syntax of the shell scripting will be discussed and multiple examples will be provided to give students solid hands-on.

The above-mentioned topics are too large to be covered in a single experiment. This is why three experiments will be dedicated to tackle shell scripting.

0.2 Objectives

The objectives of the experiment is to learn the following:

- Give a quick overview of shell scripting.
- Explain to students the used syntax, tips & tricks.
- Show students multiple examples on shell scripting.

0.3 Overview of shell scripting

Shell has its own built-in programming language. This language is *interpreted*, meaning that the shell analyzes each statement in the language one line at a time and then executes it. This differs from programming languages such as C and FORTRAN, in which the programming statements are typically compiled into a machine-executable form before they are executed[1].

Programs developed in interpreted programming languages are typically easier to debug and modify than compiled ones. However, they usually take much longer to execute than their compiled equivalents.

The shell programming language provides features you'd find in most other programming languages. It has looping constructs, decision-making statements, variables, and functions, and is procedure-oriented. Modern shells based on the IEEE POSIX standard have many other features including arrays, data typing, and built-in arithmetic operations.

As you might expect, there are several available shells and thus several ways of writing shell scripts. In this experiment, we'll be writing Bourne shell scripts, meaning shell scripts that use the `sh` shell. The main reason is that scripts your write in `sh` will run on *virtually* any shell while C-like shell scripts require `csh` shell and bash shell scripts require `bash` shell.

[1]Stephen G. Kochan, Patrick Wood - Unix Shell Programming, 3rd Edition

0.4 Command Files

You should realize that whenever you type something like:

```
who | wc -l
```

that you are actually programming in the shell! That's because the shell is interpreting the command line, recognizing the pipe symbol, connecting the output of the first command to the input of the second, and initiating execution of both commands.

A shell can be typed directly at the terminal or it can be first typed into a file and then the file can be executed by the shell.

Example

Type the following command:

```
who | wc -l
```

into file `sh1`. To execute the commands contained inside the file `sh1`, all you now have to do is type `sh1` as the command name to the shell as follows:

```
sh1
```

However, you will notice two problems that will pop up:

- You need to change the file permission of file `sh1` to make it executable. This is done with the change mode command `chmod`. To add "execute permission" to the file `sh1`, you simply type:

  ```
  chmod +x sh1
  ```

  or:

  ```
  chmod 755 sh1
  ```

- If your script `sh1` is not in your `PATH` environment variable, you need to inform the shell about the location of your script. As such, if your script `sh1` is in your current directory, you can execute it as follows:

  ```
  ./sh1
  ```

  Otherwise, you need to update the `PATH` environment variable as follows (assuming your current folder is `/home/user1/shell`):

  ```
  PATH=$PATH:/home/user1/shell
  ```

  You can now call the script by it's name as follows:

  ```
  sh1
  ```

For the next example, suppose that you want to write a shell program called `stats` that prints the date and time, the number of users logged in, and your current working directory. You know that the three command sequences you need to use to get this information are `date`, `who | wc -l`, and `pwd`.

If you execute the command `cat stats`, you would ge tthe following output:

```
date
who | wc l
pwd
```

You need afterwards to execute the command `chmod +x stats` in order to change the file permissions and make the script executable as explained above. Now run the script by calling `./stats` and make sure you get the expected output.

You might want to make the output better formated by displaying some extra messages. You can edit the script `stats` to have it contain the following lines:

```
echo "The current date and time is:"
date
echo
echo "The number of users on the system is:"
who | wc l
echo
echo "Your current working directory is:"
pwd
```

You can run the script by calling `./stats` again. You will notice that the output is more descriptive with the extra displayed messages.

0.4.1 Comments

The shell programming language would not be complete without a *comment* statement. A comment is a way for you to insert remarks or comments inside the program that otherwise have no effect on its execution.

Whenever the shell encounters the special character # at the start of a word, it takes whatever characters follow the # to the end of the line as comments and simply ignores them. If the # starts the line, the entire line is treated as a comment by the shell.

As an example, we're going to edit our script `stats` above to have it include comments as well:

```
#
# stats -- prints: date, number of users logged on,
# and current working directory
#

echo "The current date and time is:"
date
echo
echo "The number of users on the system is:"
who | wc l
echo
echo "Your current working directory is:"
pwd
```

Note that the extra blank lines cost little in terms of program space yet add much in terms of program readability. They're simply ignored by the shell.

0.4.2 Variables

Like all programming languages, the shell allows you to store values into variables. To store a value inside a shell variable, you simply write the name of the variable, followed immediately by the equals sign =, followed immediately by the value you want to store in the variable:

```
variable=value
```

Examples

```
count=1
my_path=/home/user/bin
```

Notes:

- Spaces are not permitted on either side of the equals sign. This is a major difference with other programming languages.

- Unlike most other programming languages, the shell has no concept whatsoever of *data types*. Whenever you assign a value to a shell variable, no matter what it is, the shell simply interprets that value as a string of characters. So when you assigned 1 to the variable `count` previously, the shell simply stored the *character 1* inside the variable `count`, making no observation whatsoever that an integer value was being stored in the variable.
- You can display the content of variables using the `echo` command as follows:

  ```
  echo $count
  echo $my_path
  ```

- For variables that have not been assigned values, you can still use the `echo` command to see their content. However, you will notice that the shell will not issue an error message but will display nothing. We say that a variable that contains no value is said to contain the *null* value.

  Alternatively, you can list two adjacent pairs of quotes after the =. So,

  ```
  dataflag=""
  ```

  and

  ```
  dataflag=''
  ```

  both have the same effect of assigning the null value to `dataflag`.

0.4.3 Filename Substitution and Variables

If you execute the following:

```
x=*
```

and then execute

```
echo $x
```

you will be surprised to get the list of files and folders for the location where the `echo` command was executed.

The truth is that the shell does not perform filename substitution when assigning values to variables. the substitution takes place on execution.

If you need to assign a * to the variable `x`, you need to do the following:

```
x="*"
```

and you use the `echo` command as follows:

```
echo "$x"
```

0.4.4 The ${variable} Construct

Suppose that you have the name of a file stored in the variable `filename`. If you wanted to rename that file so that the new name was the same as the old, except with an `X` added to the end, you might do the following:

```
mv $filename $filenameX
```

You will get an error message saying that the variable `filenameX` does not exist. To avoid this problem, you can delimit the end of the variable name by enclosing the entire name (but not the leading dollar sign) in a pair of curly braces, as in

```
${filename}X
```

This removes the ambiguity, and the `mv` command then works as desired:

```
mv $filename ${filename}X
```

Example

- Execute the command
  `touch filename.txt`
- Execute the command
  `var1=filename.txt`
- Execute the command
  `echo $var1`
  so as to make sure that the assignment operation succeeded.
- Execute the command
  `mv $var1 ${var1}X`
- Execute the command
  `ls`
  to make sure that `filename.txt` has been changed to `filename.txtX`

0.4.5 Built-in Integer Arithmetic

The format for arithmetic expansion is

```
$((expression))
```

where `expression` is an arithmetic expression using shell variables and operators. Valid shell variables are those that contain numeric values.

Example 1

- Set a variable `a` to 1 as follows:
  `a=1`
- Execute the following command:
  `echo $((a+1))`
- You will notice that the shell prints the value `2` on the standard output.

Example 2

- Set a variable `a` to 1 as follows:
  `a=1`
- Execute the following command:
  `result=$((a >= 0 && a <=100))`
- Execute the following command:
  `echo $result`
- You will notice that the shell prints the value `1` (which means `TRUE`) on the standard output.

The `expr` Command

Although the POSIX standard shell supports built-in integer arithmetic operations, older shells don't. It's likely that you may see command substitution with a Unix program called `expr`, which evaluates an expression given to it on the command line:

```
expr 1 + 2
```

will evaluate to 3.

Note that you should leave a blank before and after the `+` sign. Otherwise, if you execute:

```
expr 1+2
```

you might get the following output:

```
1+2
```

The usual arithmetic operators that are recognized by expr: `+` for addition, `-` for subtraction, `/` for division, `*` for multiplication, and `%` for modulus (remainder).

Example 1

Execute the following:

```
expr 10 + 20 / 2
```

and note the output that you get.

Example 2

Execute the following:

```
expr 5 * 5
```

Can you explain why you didn't get the number `25` as you expected?

The reason for the error you got is that the shell saw the `*` and substituted the names of all the files in your directory.

To avoid having that problem, execute the following:

```
expr 5 \* 5
```

and note the output that you get.

Example 3

Execute the following in sequence:

```
a=1
a=$(expr $a + 1)
echo $a
```

0.5 The use of quotes

In the section, you will learn how the shell interprets the different quotes it might encounter.

0.5.1 Single quotes

You might need single quotes to keep together characters that are separated by whitespace characters.

Example 1

Assume you have the file `addresses.txt` with the following content:

```
Salem Taweel - Jerusalem
Fadia Khalil - Bethlehem
Nabil Asmar - Ramallah
```

```
Mohammad Badie - Jericho
Salem Kaseer - Jenin
```

You can look for people in your file using the `grep` command that we've seen before as follows:

```
grep Fadia addresses.txt
```

The above command will print:

```
Fadia Khalil - Bethlehem
```

If you now execute:

```
grep Salem addresses.txt
```

you'll get the following two entries:

```
Salem Taweel - Jerusalem
Salem Kaseer - Jenin
```

If you only needed to get the entry for `Salem Kaseer`, you might be tempted to execute:

```
grep Salem Kaseer addresses.txt
```

However, you'll get an error message stating that the file `Kaseer` cannot be found:

```
grep:  Kaseer:  No such file or directory
```

The solution in to consider `Salem Kaseer` as one unit as follows:

```
grep 'Salem Kaseer' addresses.txt
```

Example 2

If you execute the command:

```
echo *
```

you'll get the list of files and folders under the current directory. If you need to just display the character `*`, you execute:

```
echo '*'
```

0.5.2 Double quotes

Double quotes work similarly to single quotes, except that they're not as restrictive. Whereas the single quotes tell the shell to ignore *all* enclosed characters, double quotes say to ignore *most*. In particular, the following three characters are not ignored inside double quotes:

- Dollar signs ($)
- Back quotes (`)
- Backslashes (\)

Example 1

- Execute the command:
  ```
  x=*
  ```
- Execute the command:
  ```
  echo $x
  ```
- Execute the command:
  ```
  echo '$x'
  ```
- Execute the command:
  ```
  echo "$x"
  ```

Note the difference in output that you get at each execution. In the first case, the shell sees the asterisk and substitutes all the filenames from the current directory. In the second case, the shell leaves the characters enclosed within the single quotes alone, which results in the display of `$x`. In the final case, the double quotes indicate to the shell that variable name substitution is still to be performed inside the quotes. So the shell substitutes `*` for `$x`. Because filename substitution is not done inside double quotes, `*` is then passed to `echo` as the value to be displayed.

0.5.3 The backslash

The backslash is equivalent to placing single quotes around a single character, with a few minor exceptions. The backslash quotes the single character that immediately follows it.

Example 1

If you run the command:

```
echo >
```

you'll get the following error:

```
Missing name for redirect.
```

or

```
syntax error:  'newline or ;' unexpected
```

Depending on the Linux distribution that you have, you might get even a different error message. The problem that you are getting here is that the shell uses the `>` character as a redirection command but is unable to find *what* to redirect and to *where* to redirect.

Example 2

Execute the following in sequence and notice the output that you get in each case:

```
x=*
echo \$x
echo '\$x'
echo "\$x"
echo \$x
echo \\
```

0.6 Command Substitution

Command substitution refers to the shell's capability to insert the standard output of a command at any point in a command line.

There are two ways in the shell to perform command substitution:

- By enclosing a shell command with back quotes (`).
- By using the `$(...)` construct.

Example 1

Execute the following command:

```
echo "The date and time is:  `date`"
```

Example 2

Execute the following command:

```
echo "Your current working directory is `pwd`"
```

Note that in the above two examples, the shell substituted the `date` and `pwd` commands by their output.

Note that we get the same behavior if we execute the following command:

```
echo "The date and time is:  $(date)"
```

Example 3

Execute the following command:

```
echo "There are $(who | wc -l) users logged in"
```

You'll notice that the shell substituted the command `who | wc -l` by its output.

Example 4

Execute the following in sequence:

```
now=$(date)
echo $now
```

and note the value of the variable `now`. You'll notice that it stores the output of the command `date` at the time of execution.

Example 5

Execute the following in sequence:

```
filelist=$(ls)
echo $filelist
```

and note the value of the variable `filelist`. Note as well that you get a slightly different output if you execute the command:

```
echo "$filelist"
```

Example 6

Execute the following in sequence:

```
name="Firstname Lastname"
name=$(echo $name | tr '[a-z]' '[A-Z]')
echo $name
```

and note the value of the variable `name` becomes `FIRSTNAME LASTNAME`.

Let's do now a slightly more complicated example where we'll show a nested command substitution:

```
filename=/home/user/exp9
filename=$(echo $filename | tr "$(echo $filename | cut -c1)" "^")
echo $filename
```

Note that you should get the following output when executing `echo $filename`:

```
^home^user^exp9
```

0.7 Passing Arguments to Shell Scripts

Shell programs become more useful when you learn how to process arguments passed to them.

Whenever you execute a shell program, the shell automatically stores the first argument in the special shell variable `$1`, the second argument in the variable `$2`, and so on up to variable `$9`. You might wonder what can we do if the number of arguments exceeds 9. Let's keep that for later.

Example 1

Consider you want to create a small script called `isOn` to which you pass the specified username as an argument. You can do it as follows. Open the script `isOn` and type the following:

```
who | grep $1
```

You can now run the script as follows:

```
./isOn sameer
```

if you want to check if user `sameer` is logged on or not. Remember to make the script `isOn` executable using the command `chmod` as explained above.

0.7.1 The $# Variable

Whenever you execute a shell program, the special shell variable `$#` gets set to the number of arguments that were typed on the command line. This variable can thus be tested by the program to determine whether the correct number of arguments was typed by the user.

Example 1

We'll open a script called `dollar_pound` and type the following in it:

```
echo $# arguments passed
echo arg 1 = :$1: arg 2 = :$2: arg 3 = :$3: arg 4 = :$4: arg 5 = :$5:
```

Try running the script as follows and note the output that you get:

```
./dollar_pound 1 2 3 4 5
```

Try as well the following scenario:

```
./dollar_pound 'user 1' 'user 2' 'user 3' 'What is your name' 'How are you'
```

Example 2

As a more complicated example using command substitution, consider that you have the file `names` that contains the following line:

```
user_1 user_2 user_3 What_is_your_name How_are_you
```

You can run the script `dollar_pound` on file `names` as follows:

```
./dollar_pound $(cat names)
```

0.7.2 The $* Variable

The special variable `$*` references *all* the arguments passed to the program. This is often useful in programs that take an indeterminate or variable number of arguments. Let's see a quick example.

Example 1

Create the script `dollar_star` and type the following:

```
echo $# arguments passed
echo they are :$*:
```

Make the script `dollar_star` executable and run it as follows:

```
./dollar_star arg1 arg2 arg3
```

Note the output that you get.

Example 2

We'll see below a more complicated example. We intend to create simple scripts that would manage a phone book document, add more users to it and delete from it if needed. You'll see that schell scripting can do a nice job and does not require lots of efforts.

Assume you have the file `addresses_phones.txt` that contains the following entries:

```
Fadia Khalil 7654321
Mohammad Badie 4433221
Nabil Asmar 1122334
Salem Kaseer 9988776
Salem Taweel 1234567
```

Create the script `phonebook` that contains the following line:

```
grep "$1" phonebook
```

Make the script `phonebook` executable. You can use it as follows:

```
./phonebook Salem
```

You should get the following two entries:

```
Salem Kaseer 9988776
Salem Taweel 1234567
```

Let's create now the script `phonebook_add` that adds new users to the file `addresses_phones.txt`.

```
#
# Add someone to the phone book
#

echo "$1 $2" >> phonebook
```

To add a new entry, you type:

```
./phonebook_add 'Fathi Khalil' 1112223
```

Check that the entry has been added to the file `addresses_phones.txt`

One problem you might notice already is that your file `addresses_phones.txt` is no more sorted. To keep the file sorted, we'll do a slight modification to the script `phonebook_add` as follows:

```
#
# Add someone to the phone book
#

echo "$1 $2" >> phonebook

sort -o phonebook phonebook
```

Add the following entry and make sure it will be placed in file `addresses_phones.txt` at the right line:

```
./phonebook_add 'Amir Nawar' 2345125
```

We now just need to be able to remove users from file `addresses_phones.txt` if needed. let's create the script `phonebook_rm` that removes users from file `addresses_phones.txt`.

```
#
# Remove someone from the phone book
#

grep -v "$1" phonebook > /tmp/phonebook

mv /tmp/phonebook phonebook
```

Note the use of the command `grep` with the `-v` option. Note as well that in order to remove a particular user, you need to do it in 2 steps as shown above. Can you explain why?

For those who do not know already, the `/tmp` directory exists on all Unix/Linux systems and anyone can write to it. It is used by programs to create *temporary* files. Each time the system gets rebooted, all the files in /tmp are usually removed. Users thus should not store any file under `/tmp` unless they do not care about.

0.7.3 Passing more than 9 arguments to shell scripts

If you supply more than nine arguments to a program, you cannot access the tenth and greater arguments with `$10`, `$11`, and so on. If you try to access the tenth argument by writing:

```
$10
```

the shell actually substitutes the value of `$1` followed by a `0`.

In order to access these high-order arguments, you need to enclose them in {}. For example, in order to access the 11^{th} argument, the script should do as follows:

```
${11}
```

0.7.4 The `shift` command

The `shift` command allows you to effectively left shift your positional parameters. If you execute the command:

```
shift
```

whatever was previously stored inside `$2` will be assigned to `$1`, whatever was previously stored in `$3` will be assigned to `$2`, and so on. The old value of `$1` will be lost. In addition, `$#` (the number of arguments variable) is also automatically decremented by one.

Example 1

Create the script `tshift` and type the following:

```
echo $# $*
shift

echo $# $*
shift

echo $# $*
shift

echo $# $*
shift

echo $# $*
shift

echo $# $*
```

Make the script `tshift` executable and run it as follows:

```
tshift a b c d e
```

Note the output that you get.

Note as well that the command `shift` takes an optional argument. If that argument is absent, `shift` assumes a default of 1. Otherwise, it will execute a number of shifts equivalent to the argument.

Example 2

Create the script `tshift3` and type the following:

```
echo $# $*
shift 3

echo $# $*
shift 3

echo $# $*
shift 3
```

Make the script `tshift3` executable and run it as follows:

```
tshift3 aa bb cc dd ee ff gg hh ii
```

Note the output that you get.

0.8 Exit Status

Whenever any program completes execution under the Unix/Linux systems, it returns an exit status back to the system. This status is a number that usually indicates whether the program successfully ran. By convention, an exit status of zero indicates that a program succeeded, and nonzero indicates that it failed.

Failures can be caused by invalid arguments passed to the program, or by an error condition detected by the program.

Shell scripts can use the exist status of commands to decide which way to go in a script. For example, if the execution of the previous command succeeds, a script can decide to do action 1 or else executes action 2 in case of failure.

0.8.1 The `$?` variable

The shell variable `$?` is automatically set by the shell to the exit status of the last command executed. Naturally, you can use the `echo` command to display its value at the terminal.

Example 1

Execute the following in sequence:

```
touch fileNoExist
echo $?
ls -l fileNoExist
echo $?
\rm fileNoExist
echo $?
\rm fileNoExist
echo $?
```

Note that `echo $?` would display the value `0` the first 3 times it is called but would display a *positive* value on the fourth execution. The reason is that the execution of commands `touch fileNoExist`, `ls -l fileNoExist` and `\rm fileNoExist` succeeded. However, when you try to remove a file that does not exist, the command fails and generates a positive exit status.

Example 2

Execute the following in sequence:

```
pwd
echo $?
pwdd
echo $?
```

Note that the first execution of `echo $?` would display the value `0` while the second execution would display a positive value.

Example 3

Execute the following in sequence:

```
cd /
echo $?
touch fileNoExist
echo $?
```

You will notice here that the first execution of `echo $?` would display the value `0` while the second execution would display a positive value since you are not allowed to create the file `fileNoExist` in a directory to which you have no write-access (unless you are logged to the system with an admin account).