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

Experiment #8
Shell scripting
Part II

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 scripts that take advantage of the `test` command, will be shown string operators, integer operators, file operators, logical AND operator and logical OR operator. Students will be shown also the `case` command, the syntax of `if...else...fi` command and the Null command. Students will finally be exposed to debugging shell scripts with the `-x` option[1].

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:

- Provide more insight into the syntax shell scripting.
- Show students multiple examples on shell scripting.

0.3 The $$ variable

The special variable `$$` contains the process id number (PID) of the current process.

Example 1

Type on the shell the following command:

```
echo $$
```

and check on the output that you get. Note that the displayed PID is that of your shell. To verify that, execute the command `ps` and look for the process that holds the PID you just got.

Example 2

Execute the following in sequence:

```
touch coco.txt
mv coco.txt coco.$$_$$
```

Check on the new name that was given to the file `coco.txt`.

Note

You might be wondering about how useful is the special variable $$. The answer is as follows:

- There are cases where you might need to know your PID as a running shell script. Multiple actions can be taken in such cases (e.g. kill the script).

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

- It might happen that your shell script needs to create temporary files while it is running. In order for your shell script to succeed in its mission, these temporary files need not exist when you are about to create them. One way would be to append the special variable $$ to these temporary files (or directories) so as to make sure they are unique:

  ```
  touch coco.txt.$$
  ```

  The above becomes even more critical in a multiprocessing environment, where users might concurrently be writing temporary files to the same location (e.g. `/tmp` folder).

0.4 The `test` Command

A built-in shell command called `test` is most often used for testing one or more conditions. Its general format is

```
test expression
```

Where `expression` represents the condition you're testing. `test` evaluates `expression`, and if the result is *true*, it returns an exit status of zero; otherwise, the result is *false* and it returns a nonzero exit status.

0.4.1 String Operators

Example 1

Execute the following in sequence:

```
name="user"
test $name = "user"
echo $?
test $name = "userr"
echo $?
```

The `=` operator is used to test whether two values are identical. If they are, `test` returns an exit status of zero; nonzero otherwise.

Note that while using the `test` command, you must delimit the `=` sign by whitespace characters (before and after the `=` sign).

Note

It is good programming practice to enclose shell variables that are arguments to `test` command inside a pair of double quotes (to allow variable substitution). This ensures that `test` sees the argument in the case where its value is null. For example, execute the following in sequence:

```
name=
test $name = "user"
echo $?
```

You will get an error message saying that command `test` is expecting an argument. The reason is that the shell substituted `$name` by null so the substitution looked like:

```
test = "users"
```

which is a wrong usage of the command `test`.

To avoid this problem, you can place double quotes around the variable `$name` as follows:

```
name=
test "$name" = "user"
echo $?
```

To test character strings, other operators can be used. These operators are summarized in the following table:

Operator	Returns TRUE (exit status of 0) if
`string`$_1$ `= string`$_2$	`string`$_1$ is identical to `string`$_2$
`string`$_1$ `!= string`$_2$	`string`$_1$ is not identical to `string`$_2$
`string`	`string` is not null
`-n string`	`string` is not null (and `string` must be seen by `test`)
`-z string`	`string` is null (and `string` must be seen by `test`)

Example

Execute the following in sequence:

```
name=
test -n "$name"
echo $?
test -z "$name"
echo $?
```

You will notice that the first execution of `echo $?` will generate a positive number since the previous command failed (variable `$name` is indeed null) while the second execution of `echo $?` succeeded.

0.4.2 An Alternative Format for `test`

The `test` command has an alternative format that can be used by programmers. The format is:

```
[ expression ]
```

The above format is exactly equivalent to:

```
test expression
```

Surprisingly, programmers like using brackets `[]` instead of text. This is why the first format is more popular than the second.

Example

Execute the following in sequence:

```
name=
[ -n "$name" ]
echo $?
[ -z "$name" ]
echo $?
```

Note that the results you get in this example are equivalent to the ones that you got in the previous example.

0.4.3 Integer Operators

In addition to the string operators, command `test` (or `[...]`) has integer operators as well. The table below summarizes these operators:

Operator	Returns TRUE (exit status of 0) if
`int1 -eq int2`	`int1` is equal to `int2`
`int1 -ge int2`	`int1` is greater than or equal to `int2`
`int1 -gt int2`	`int1` is greater than `int2`
`int1 -le int2`	`int1` is less than or equal to `int2`
`int1 -lt int2`	`int1` is less than `int2`
`int1 -ne int2`	`int1` is not equal to `int2`

Example 1

Consider the following example:

```
count=3
[ "$count" -eq 0 ]
echo $?
[ "$count" -lt 5 ]
echo $?
[ "$count" -le 3 ]
echo $?
```

You will notice that the command `echo $?` will display a positive number (previous command failed) while the other two executions of `echo $?` output a 0 (previous command succeeded).

Example 2

Consider the following example:

```
x1="005"
x2=" 10"
[ "$x1" = 5 ]
echo $?
[ "$x1" -eq 5 ]
echo $?
[ "$x2" = 10 ]
echo $?
[ "$x2" -eq 10 ]
echo $?
```

You will notice that first execution of `echo $?` will display a positive number since the string comparison of `$x1` to number `5` fails. However, the second execution of `echo $?` outputs a `0` since the integer comparison succeeds. The third execution of `echo $?` will display a positive number since the string comparison of `$x2` to number `10` fails. The fourth `echo $?` outputs a `0` since the integer comparison succeeds.

0.4.4 File Operators

The shell provides many operators that deal with files. These operators are *unary* in nature, meaning that they expect a single argument to follow. The table below summarizes these operators:

Operator	Returns TRUE (exit status of 0) if
`-d file`	`file` is a directory
`-e file`	`file` exists
`-f file`	`file` is an ordinary file
`-r file`	`file` is readable by the process
`-s file`	`file` has nonzero length
`-w file`	`file` is writable by the process
`-x file`	`file` is executable
`-L file`	`file` is a symbolic link

Example 1

Execute the following in sequence:

```
touch coco.txt
[ ! -f coco.txt ]
echo $?
[ -f coco.txt ]
echo $?
[ -d coco.txt ]
echo $?
mkdir myDir
[ -d myDir ]
echo $?
[ ! -d myDir ]
echo $?
```

You will notice that the first execution of `echo $?` will display a positive number since `coco.txt` is a file while the second execution of `echo $?` outputs a `0`. The third `echo $?` outputs a positive number since `coco.txt` is not a directory. The fourth `echo $?` outputs a `0` since `myDir` is a directory while the fifth `echo $?` fails for the same reason.

0.5 The Logical AND Operator `-a`

The operator `-a` performs a logical AND of two expressions and returns true only if the two joined expressions are both true.

Example

Execute the following in sequence:

```
count=3
[ "$count" -ge 0 -a "$count" -lt 10 ]
echo $?
```

The statement `[ "$count" -ge 0 -a "$count" -lt 10 ]` will be true if the variable `count` contains an integer value greater than or equal to zero but less than 10. As such, the `echo $?`

will display a `0` since the previous expression succeeds.

Note that The `-a` operator has lower *precedence* than the integer comparison operators (and the string and file operators), meaning that the preceding expression gets evaluated as:

```
[ ("$count" -ge 0) -a ("$count" -lt 10) ]
```

Watch out that the parantheses have a special meaning to the shell. Thus if you need to use them, they have to be quoted. The correct syntax is thus as follows:

```
[ \( "$count" -ge 0 \) -a \( "$count" -lt 10 \) ]
```

0.6 The Logical OR Operator `-o`

The `-o` operator is similar to the `-a` operator, only it forms a logical OR of two expressions. That is, evaluation of the expression will be true if *either* the first expression is true or the second expression is true.

Example

Execute the following in sequence:

```
count=3
[ "$count" -ge 0 -o "$count" -lt 10 ]
echo $?
```

The `echo $?` will display a `0` since the previous expression succeeds.

The `-o` operator has lower precedence than the `-a` operator, meaning that the expression

```
"$a" -eq 0 -o "$b" -eq 2 -a "$c" -eq 10
```

gets evaluated by `test` as:

```
"$a" -eq 0 -o ("$b" -eq 2 -a "$c" -eq 10)
```

Remember though that you need to quote the parantheses as follows:

```
\( "$a" -eq 0 -o "$b" -eq 2 \) -a "$c" -eq 10
```

0.7 `if...else...fi`

It is very useful to be able to branch in your shell scripts so that the execution path changes according to the encountered situation. If a condition is evaluated to TRUE then the shell executes a set of actions and if it evaluated to FALSE then the shell executes another set of actions. The syntax is as follows:

```
if commandt
then
     command
     command
     ...
else
     command
     command
     ...
fi
```

Execution goes on as follows: `commandt` is executed and its exit status tested. If it's zero, the commands that follow between the `then` and the `else` are executed, and the commands between the `else` and `fi` are skipped. Otherwise, the exit status is nonzero and the commands between the `then` and `else` are skipped and the commands between the `else` and `fi` are executed. In either case, only one set of commands gets executed: the first set if the exit status is zero, and the second set if it's nonzero.

Example 1

Consider the following example:

```
#!/bin/sh

#
# An example on if-else operators
#

#
# determine if someone is logged on
#

user="$1"

if who | grep "^$user " > /dev/null
then
    echo "$user is logged on"
else
    echo "$user is not logged on"
fi
```

The above example checks if a particular user (passed as an argument to the shell script) is logged on or not. Certain notes need to be done at this stage:

- The shell script starts with the line `!/bin/sh`. This informs the system that a Bourne shell should run the below script.
- The `if` and `then` constructs need to be on separate lines. If you want to keep them on the same line so that your shell script doesn't get too long, you need to replace:

  ```
  if who | grep "^$user " > /dev/null
  then
  ```

  by

  ```
  if [ who | grep "^$user " > /dev/null ] ; then
  ```

- If a user exists and is logged on, the command `grep` will have its own output showing the shells on which the user is currently logged on. In order to dispose `grep`'s output, we can redirect it to the system's *garbage can* (`/dev/null`). This is a special file on the system that anyone can read from (and get an immediate end of file) or write to.

A nice thing to do when writing shell programs is to make sure that the correct number of arguments is passed to the program. If an incorrect number is supplied, an error message can be displayed, together with information on the proper usage of the program. As such, the above example can be written in a more robust way as follows:

Example 2

```
#!/bin/sh

#
# An example on if-else operators
#

#
# see if the correct number of arguments were supplied
#

if [ "$#" -ne 1 ]
then
    echo "Incorrect number of arguments"
    echo "Usage: only one argument is required!"
else
    user="$1"

    if who | grep "^$user " > /dev/null
    then
        echo "$user is logged on"
    else
        echo "$user is not logged on"
    fi
fi
```

In the above example, if a user supplies by mistake the login name of two users, the shell script will issue an error meesage. Otherwise, the script will proceed until completion.

0.8 The `exit` Command

A built-in shell command called `exit` enables you to immediately terminate execution of your shell program. The general format of this command is

```
exit n
```

where `n` is the exit status that you want returned. If none is specified, the exit status used is that of the last command executed before the `exit`.

0.9 The `elif` Construct

As your shell scripts become more complex, you can avoid having to write nested `if` statements by using the `elif` construct (similar to the `else if` construct in C-language). Let's see how to use it in a simple example:

Example 1

Create a file called `greetings1` and type the following:

```
#!/bin/sh

#
# An example on if-elif-else operators
#

#
# Program to print a greeting
```

```
#

hour=$(date | cut -c12-13)

if [ "$hour" -ge 0 -a "$hour" -le 11 ]
then
    echo "Good morning"
else
    if [ "$hour" -ge 12 -a "$hour" -le 17 ]
    then
echo "Good afternoon"
    else
echo "Good evening"
    fi
fi
```

The shell script above will display the message `Good morning`, `Good afternoon` or `Good evening` depending on the time. Note the existence of nested `if` constructs in the above example.

We'll write below the same example using the `elif` construct.

Example 2

Create a file called `greetings2` and type the following:

```
#!/bin/sh

#
# An example on if-elif-else operators
#

#
# Program to print a greeting
#

hour=$(date | cut -c12-13)

if [ "$hour" -ge 0 -a "$hour" -le 11 ]
then
    echo "Good morning"
elif [ "$hour" -ge 12 -a "$hour" -le 17 ]
then
    echo "Good afternoon"
else
    echo "Good evening"
fi
```

This version is easier to read, and it doesn't have the tendency to disappear off the right margin due to excessive indentation:-)

0.10 The `case` Command

The `case` command allows a user to compare a single value against other values and to execute one or more commands when a match is found. The general format of this command is:

```
case value in
pat1) command
```

```
      command
      ...
      command;;

pat2) command
      command
      ...
      command;;
...

patn) command
      command
      ...
      command;;
esac
```

You will notice a high resemblance between the `case` command and the `switch...case` in C-language.

In the above format, the word `value` is successively compared against the values `pat1`, `pat2`, ..., `patn` until a match is found. When a match is found, the commands listed after the matching value, up to the double semicolons, are executed. After the double semicolons are reached, execution of the case is terminated. If a match is not found, none of the commands listed in the case is executed.

Example 1

Create a file called `case_1` and type the following:

```
#!/bin/sh

#
# An example on case operators
#

#
# Translate a digit to English
#

if [ "$#" -ne 1 ]
then
    echo "Usage: number digit"
    exit 1
fi

case "$1"
    in
    0) echo zero;;
    1) echo one;;
    2) echo two;;
    3) echo three;;
    4) echo four;;
    5) echo five;;
    6) echo six;;
    7) echo seven;;
    8) echo eight;;
```

```
    9) echo nine;;
esac
```

The above example will translate numbers in the range `[0...9]` into their respective string. To test it, run the command:

```
./case_1 0
```

and check on the output that you get.

0.10.1 Special Pattern Matching Characters

The shell lets you use the same special characters for specifying the patterns in a case as you can with filename substitution. That is, `?` can be used to specify any single character; `*` can be used to specify zero or more occurrences of any character; and `[...]` can be used to specify any single character enclosed between the brackets.

The `*` in that case behaves as the `default` clause in a `switch...case` in C-language.

Example 2

The above example can be re-written as below. Create a file called `case_2` and type the following:

```
#!/bin/sh

#
# An example on case operators
#

#
# Translate a digit to English
#

if [ "$#" -ne 1 ]
then
    echo "Usage: number digit"
    exit 1
fi

case "$1"
    in
    0) echo zero;;
    1) echo one;;
    2) echo two;;
    3) echo three;;
    4) echo four;;
    5) echo five;;
    6) echo six;;
    7) echo seven;;
    8) echo eight;;
    9) echo nine;;
    *) echo "Bad argument; please specify a single digit";;
esac
```

Example 3

As an additional example on the use of the `case` command, let's consider a script that prints the type of the single character given as an argument. Character types recognized are digits, uppercase letters, lowercase letters, and special characters (anything not in the first three categories).

Create a file called `case_3` and type the following:

```
#!/bin/sh

#
# An example on case operators
#

#
# Classify character given as argument
#

if [ $# -ne 1 ]
then
    echo Usage: case_3 char
    exit 1
fi

#
# Ensure that only one character was typed
#
char="$1"
numchars=$(echo -n "$char" | wc -c)

if [ "$numchars" -ne 1 ]
then
    echo Please type a single character
    exit 1
fi

#
# Now classify it
#

case "$char"
    in
    [0-9] ) echo digit;;
    [a-z] ) echo lowercase letter;;
    [A-Z] ) echo uppercase letter;;
    * ) echo special character;;
esac
```

To test the above script, run the command in the below sequence and make sure you get the expected output after each execution:

```
./case_3 a
./case_3 Z
./case_3 3
./case_3 @
./case_3 123
```

0.11 The Null Command :

The purpose of the command `:` is to do nothing! This is a little bit weird. However, it is mainly used after an `if` command succeeds but you have nothing to do in such a case. Let's illustrate

how to use it in an example.

Example

Assume you have the following script:

```
if who | grep "^$user " > /dev/null
then
    :
else
    echo "$user is not logged on"
fi
```

In the above example, if the sentence `who | grep "^$user " > /dev/null` succeeds, you do not want to do anything special. Unfortunately, the shell requires that you write a command after the `then` construct. Here is where the null command becomes useful. You just tell the shell to do nothing in such a case.

If the sentence `who | grep "^$user " > /dev/null` fails, the commands between the `else` and `fi` get executed.

Remember this simple command when these types of situations arise.

0.12 The && and || Constructs

The shell has two special constructs that enable you to execute a command based on whether the preceding command succeeds or fails. This is a shortcut for the `if` command. The syntax is as follows:

```
commandl && command2
```

The above syntax means that if `command1` gets executed and if it returns an exit status of zero (success), `command2` will be executed. If `command1` returns an exit status of nonzero (failure), `command2` gets skipped.

Example

Create a text file called bigdata and store the following in it:

```
10
2
34
22
```

Execute then the following command:

```
sort bigdata > /tmp/sortout && mv /tmp/sortout bigdata
```

In the above command, the `mv` command will be executed *only* if the `sort` is successful. The above command is equivalent to:

```
if sort bigdata > /tmp/sortout
then
    mv /tmp/sortout bigdata
fi
```

The `||` construct works similarly, except that the second command gets executed only if the exit status of the first is nonzero (failure). So if you write:

```
grep "$name" phonebook || echo "Couldn't find $name"
```

the `echo` command will get executed only if the `grep` fails (that is, if it can't find `$name` in `phonebook`, or if it can't open the file `phonebook`). In this case, the equivalent `if` command would look like

```
if grep "$name" phonebook
then
    :
else
    echo "Couldn't find $name"
fi
```

0.13 Debugging shell scripts with the -x option

We never write programs that run without errors from the first shot unless they are composed of few lines. Useful applications are composed of a huge number of lines of code. Debugging becomes tedious with every additional line of code.

High level programming languages have all debuggers that enable users to verify variable content, memory content, break at a particular instruction, etc. An example is the `gdb` debugger we've seen earlier

Unfotunately, the shell doesn't have all that luxury. It just has a facility to trace a script during execution and thus know the lines that have been stepped into before completing the job. By examining the script trace, we can know if it executed right or not.

You can trace the execution of any program by typing `sh -x` followed by the name of the program and its arguments. This starts up a new shell to execute the indicated program with the `-x` option enabled. In this mode, commands are printed at the terminal as they are executed, preceded by a plus sign (`+`).

Example

The example below has been previously seen (script `case3`). Only the comments have been removed to make it shorter. We intend to run it with the trace option `-x`.

```
#!/bin/sh

if [ $# -ne 1 ]
then
    echo Usage: case_3 char
    exit 1
fi

char="$1"
numchars=$(echo -n "$char" | wc -c)

if [ "$numchars" -ne 1 ]
then
    echo Please type a single character
    exit 1
fi

case "$char"
    in
    [0-9] ) echo digit;;
    [a-z] ) echo lowercase letter;;
    [A-Z] ) echo uppercase letter;;
```

```
    * ) echo special character;;
esac
```

We'll run it with the `-x` option as follows:

```
sh -x ./case_3 a
```

You will get the below output:

```
+ [ 1 -ne 1 ]
+ char=a
+ echo a
+ wc c
+ numchars= 2
+ [ 2 -ne 1 ]
+ echo please type a single character
please type a single character
+ exit 1
```

In the above output, you can see each line that the shell has executed in the script `case_3` and the values that were substituted to the different variables. You can thus detect any anomaly or situation that you did not account for. For example, the user passed argument `a` and thus you would assume that the command `wc -c` would return 1. However, you notice that the variable `numchars` has the value of 2. Why?

The reason is that the character `a` is followed by the invisible newline character that `echo` automatically prints at the end of each line. So the program really should be testing for the number of characters equal to two: the character typed plus the newline added by `echo`.