UT 013.
INTRODUCTION TO SHELL SCRIPTING

Computer Systems
CFGS DAW

Álvaro Maceda

a.macedaarranz@edu.gva.es

2022/2023

Version:230324.1242

License

Nomenclature

Throughout this unit different symbols will be used to distinguish important elements within the content. These symbols are:

TABLE OF CONTENTS

1. What is a shell script?.....4
1.1 An example of a shell script.....4
2. Commands.....5
2.1 How a command is interpreted.....5
2.2 Variables.....7
2.3 Arithmetic expressions.....8
2.4 Quotes.....8
3. Control structures.....10
3.1 Conditions.....10
3.2 Conditional sentences.....13
4. Supplementary material.....15

UT 013. Introduction to shell scripting

1. What is a shell script?

A shell script is a program meant to be run by a shell. Remember that the shell is the "place" where we run orders in the console, so the are used to automatize things that we will usually do using the console.

Shell scripts are composed of ASCII text and can be created using a text editor, word processor, or graphical user interface. These scripts consist of a sequence of commands in a language that the shell can interpret. Shell scripts are capable of supporting various functions including loops, variables, if/then/else statements, arrays, and shortcuts. Once the script is finalized, it is saved typically with .sh extension.

In this course, we will work with `bash` shell scripts. Other shells, like `zsh`, have other capabilities and the scripts written for a shell won't always run in another shell.

1.1 An example of a shell script

This is a simple example of a shell script. Don't worry if you don't understand how it works yet:

```
#!/usr/bin/env bash

# Initialize directory counter to zero
dir_count=0

# Loop through the files in the current directory
for file in *
do
  # Check if the file is a directory
  if [ -d "$file" ]
  then
    echo "$file is a directory"
    ((dir_count++)) # increment the directory count
  else
    echo "$file is not a directory"
  fi
done

# Print the file and directory counts
echo "Found $file_count files and $dir_count directories"
```

The script starts with a special line, the shebang. After that, you have comments (text after the `#` symbol) commands (like `echo`) control structures (`for`, `do`, `in`, `if`, `else`...) and variables (`dir_count`).

To run the script, you need to assign the file execute permissions and launch it from the shell. Assuming it's named `simple_example.sh`:

```
./simple_example.sh
```

One important thing is that **the shell script will also work if we write the same text directly into the terminal**, because what we do with a shell script is to store in a file a series of commands so we don't need to copy them again and again when trying to do something in the system.

1.1.1 shebang

The shebang (also called a hashbang or a pound bang) is the first line in a script file that starts with the characters `#!` (hash and exclamation mark). It is used to specify the interpreter or shell that should be used to run the script. When a script file is executed, the operating system reads the shebang line to determine the interpreter or shell that should be used to execute the commands in the script. For example, a shebang line of `#!/bin/bash` would indicate that the Bash shell should be used to interpret and execute the commands in the script.

You should use `#!/usr/bin/env bash` for portability: different *nixes put bash in different places, and using `/usr/bin/env` is a workaround to run the first `bash` found on the PATH.

2. Commands

The main part of a shell script are the commands that we want to run. They are the same commands that you would use in a console to perform operations. Nevertheless, to use them properly in a script you need to know a little bit more about how they are processed by the shell.

2.1 How a command is interpreted

The shell doesn't run the command as you enter it in the command line This happens not only for scripts but also for the command we enter directly in a shell.

The steps to run a command are something like this:

1. Split the command into tokens
2. Replace variables with their values
3. Performs command substitution
4. Carry out wildcard expansion
5. Execute the command

We will see what happens when we try to run a command like this:

```
ls -l      $HOME/* |      grep "$(date +"%b %e")"
```

2.1.1 Split int tokens

The first step is to split the command into separate "words" at the whitespace characters. The shell ignores the amount of whitespace (spaces and TABs) between different words in a command line. Any unquoted whitespace creates a new word.

In this case, the shell will split the command and see it as a list of tokens like this:

```
ls, -l, $HOME/*, |, grep, "$(date +"%b %e")"
```

Note that the characters inside the quotes are not split. The above command, then, will be equivalent to this:

```
ls -l $HOME/* | grep "$(date +"%b %e")"
```

The shell also notices the `|` and will run two different commands setting up a pipe between them. That is a transparent process for us.

2.1.2 Replace variables

The next step is to replace the variables with their values. In a shell, we can have variables that hold some data. Variables are indicated with a preceding `$` symbol. In the above example, `$HOME` is a variable and it will be substituted with its value. Assuming that the variable `$HOME` had the value `/home/user`, The command at this point looks like this:

```
ls -l /home/user/* | grep "$(date +"%b %e")"
```

2.1.3 Command substitution

Command substitution is a feature that allows the output of a command to be used as input for another command or to be assigned to a variable. Command substitution is achieved by enclosing the command to be executed within `$(command)` or backticks: `` `command` ``. Both syntaxes are almost equivalent, but `$()` syntax is usually preferred for readability.

In the above example, we have one command substitution: `$(date +"%b %e")`. It will be replaced by the result of executing that command. If we execute that in the console we would get, for example:

```
echo $(date +"%b %e") # The echo command is for visualizing the result

mar 20 # This will be the current date
```

So the expression `$(date +"%b %e")` will be replaced with mar 20. At this point, the command looks like this:

```
ls -l /home/user/* | grep "mar 20"
```

Note that the quotes “outside” the `$()` expression are not substituted.

2.1.4 Wildcard expansion

In Linux, wildcard expansion refers to the process of using special characters to build glob patterns. Those patterns represent multiple filenames or directories at once.

The most commonly used wildcard characters are:

- `*`: Represents zero or more characters. For example, `ls *.txt` will list all files with a `.txt` extension in the current directory.
- `?` (question mark): Represents a single character. For example, `ls file?.txt` will list files named `file1.txt`, `file2.txt`, etc. but not `file10.txt`.
- `[]` (square brackets): Represents a range of characters or a set of characters. For example, `ls file[1-3].txt` will list files named `file1.txt`, `file2.txt`, and `file3.txt`, but not `file4.txt`

In the example, we have a wildcard character `*`. Assuming that in `/home/user` there are three files `file1`, `file2` and `file3`, the expansion will take place and the command will be like this:

```
ls -l /home/user/file1 /home/user/file2 /home/user/file3 | grep "mar 20"
```

2.1.5 Execute the command

Once we have the final version of the command, the shell will execute it. We will have the same result as if we have entered the initial (or any intermediate version) of the command.

2.2 Variables

In Bash, a variable is a name that represents a value. Variables are used to store data that can be used by scripts, programs, or commands.

To create a variable in Bash, you simply give it a name and assign a value to it. To read the value of a variable, you prepend a `$` to the variable name. The basic syntax is:

```
A=value
B=$A
echo $A,$B

value,value
```

Bash differs from many other programming languages in that it does not categorize its variables by "type." In Bash, variables are essentially character strings, but their ability to undergo arithmetic operations and comparisons is context-dependent. If a variable's value consists solely of digits, Bash permits such operations and comparisons to be performed.

2.2.1 Local and environment variables

In Bash, there are two types of variables: local and environment. Local variables are only available within the current shell or script, while environment variables are available to any child process that is created from the current shell. To create an environment variable, you use the export command.

For example, if we have `script1.sh`:

```
#!/bin/bash
my_var="Hello, World!"

./script2.sh
```

And `script2.sh`:

```
#!/bin/bash
echo $my_var
```

If we run `script1.sh` it will output an empty string because `$my_var` doesn't exist in `script2.sh`. If we add an `export $my_var` to `script1.sh` before calling `script2.sh`, it will output `"Hello, World!"`.

2.3 Arithmetic expressions

In shell scripts, there is a way of performing mathematical operations on integer values. Bash provides several arithmetic operators such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), modulus (`%`), and exponentiation (`**`), which can be used to perform basic mathematical operations.

To perform a numeric evaluation in shell scripts, you need to enclose the mathematical expression inside double parentheses `$((expression))`. For example, to add two numbers in shell scripts, you can use the following syntax:

```
a=5
b=10
c=$((a + b))
echo $c

15
```

The `let` command is another way of performing arithmetic operations in shell scripts. It allows you to evaluate arithmetic expressions and store the result in a variable.

```
a=5
b=10
let c=a+b
echo $c

15
```

The `let` command can be used with various arithmetic operators such as `+`, `-`, `*`, `/`, `%`, `**`, and `+=`, `-=` etc. Here are some examples:

```
a=5
b=10
let c=a+b       # addition
let d=a-b       # subtraction
let e=a*b       # multiplication
let f=a/b       # division
let g=a%b       # modulus
let h=a**b      # exponentiation
let a+=5        # increment by 5
let b-=5        # decrement by 5
```

2.4 Quotes

In shell scripting, quotes are used to control how the shell interprets and expands variables, glob patterns, and other special characters in commands and scripts. There are three types of quotes in shell scripting: one of them is the backticks (`` ` ``) used in command substitution. The other two are single quotes and double quotes.

2.4.1 Single quotes

Single quotes (`'`) preserve the literal value of all characters within them. That means that variables, glob patterns, and other special characters within single quotes are treated as plain text and not expanded or evaluated by the shell. For example:

```
echo 'Hello $USER'

Hello $USER
```

Note that the variable `$USER` is not expanded because it is within single quotes.

2.4.2 Double quotes

Double quotes (`"`)allow certain special characters within them to be expanded or evaluated by the shell. Specifically, variables within double quotes are replaced with their values, and some special characters like `$`, `*`, and `?` are expanded to match filenames and perform other functions. For example:

```
echo "Hello $USER"

Hello user1
```

Note that the variable `$USER` is expanded to your username.

2.4.3 Nesting quotes

What do you think will be the output of this command?

```
echo ""Hello""
```

Quotes in the shell don't work like in other programming languages. In shell programming, they work "enabling" and "disabling" the quotes mode. When we write a quote we enter the "quotes mode", and it will be enabled until the shell finds another quote. So, in the above example, it will happen something like this:

```
echo "(enters quotes mode)"(exits quotes mode)Hello"(enters quotes
mode)"(exits quotes mode)
```

There is no text inside the "quotes mode", so the quotes have no effect.

If we want to print a quote, we can use the escape character `\`. The character following `\` won't be interpreted as a special character:

```
echo \"Hello\"

"Hello"
```

An also:

```
echo \$HOME

$HOME
```

Quotes can also be nested within each other to control how the shell expands and evaluates commands and variables. For example:

```
echo "Hello '$USER'"

Hello 'user1'
```

Note that the single quotes within the double quotes preserve the literal value of the $USER variable, which is then expanded outside of the single quotes.

3. Control structures

To use the full power of shell programming, as in other languages, we need to use control structures. In this course, we will look at the simplest ones: conditional statements and loops. To use them, we will first need to see how to express conditions in bash.

3.1 Conditions

In Bash, a condition is an expression that can be evaluated as true or false, also known as Boolean values. Conditions are used to control the flow of a script and to perform certain actions only if a certain condition is met.

There are two ways of expressing conditions in bash: single-bracket syntax and double-bracket syntax.

3.1.1 Single-bracket syntax

It is the oldest supported syntax. It's the equivalent of running the `test` command in the console: that command will return `0` if the condition is met, and a different value if not.

With single-bracket syntax you can:

- Perform arithmetic comparisons. For example, this will be true: `[ 1 -lt 5]`
- Compare strings. For example, `"$VAR_1" == "potato"` will check if potato is the content of the variable `VAR_1`.
- Check conditions on files. For example, `[ -d  path ]` will check if path is a directory

In Linux, commands return `0` if they succeed and any other value if not. So, if the conditions are true, the exit of the command will be `0`.

You can't use `<` or `>` inside single-bracket syntax, because it will be interpreted as redirection.

A table below provides a summary of the most commonly used conditions.

3.1.2 Double-bracket syntax

This syntax is a more powerful version of the single-brackets syntax, but is less portable than the single-brackets syntax (it won't work in other shells) Allows for advanced string comparisons and pattern matching. It also allows using `<` and `>` for comparison (but not `<=` or `>=`)

It has other differences from the single-bracket syntax:

- Expressions can be grouped within double brackets using parentheses, which can help make complex expressions easier to read. For example: `[[ 3 == 3 && (2 == 2 && 1 == 1) ]]`
- The double brackets also support pattern matching, where the wildcard `*` can be used to match any characters in a string. For example, to check if a variable contains a c we can use `[[ $name = *c* ]]`
- Bash does not perform word splitting inside double brackets. For example, if a variable has a string value containing spaces, Bash will not split the string into words. This can be useful in cases where you want to test a variable that contains a full file path or other complex string.

For example, if `FILENAME="nonexistent file"` we can use `[[ ! -e $filename ]]`. That will generate an error in single-brackets syntax. It also has better support for empty variables.

3.1.3 Boolean operators

Boolean operators are used to combine or modify conditions in logical expressions. The three Boolean operators available in Bash are NOT, AND and OR.

The `-a` operator is used for performing an AND operation:

```
[[ 1 > 2 -a 1 < 3 ]] # False
[[ 1 < 2 -a 1 < 3 ]] # True

[ 1 -gt 2 -a 1 -lt 3 ] # False
[ 1 -lt 2 -a 1 -lt 3 ] # True
```

The `-o` is used for OR operations:

```
[[ 1 > 2 -o 1 < 3 ]] # True
[[ 1 > 2 -o 1 > 3 ]] # False

[ 1 -gt 2 -o 1 -lt 3 ] # True
[ 1 -gt 2 -o 1 -gt 3 ] # False
```

You can also implement AND and OR with the `&&` and `||` shell operators (we will talk about them later) Note that, for using these operators, you need to close the comparison instead of including them inside the comparison if you are using the single-bracket syntax:

```
# AND
[ 1 -gt 2 ] && [ 1 -lt 3 ] # False
[[ 1 > 2 && 1 < 3 ]]       # False

# OR
[ 1 -gt 2 ] || [ 1 -lt 3 ] # True
[[ 1 > 2 || 1 < 3 ]]       # True
```

The not operator is represented by the exclamation mark `!`:

```
[[ 1 < 5 ]] # true
! [[ 1 < 5 ]] # false
[[ ! 1 < 5 ]] # false
! [[ ! 1 < 5  ||  1 > 3 ]] # True

[ 1 -lt 5 ] # true
! [ 1 -lt 5 ] # false
[ ! 1 -lt 5 ] # false
! [ ! 1 -lt 5  -o  1 -gt 3 ] # True
```

3.1.4 Basic rules for conditions

There are certain rules that will help you to write conditions properly:

- Maintain spaces between the brackets and the actual comparison

  Bad:

  ```
  [1 -lt 2]
  ```

Good:

```
[ 1 -lt 2 ]
```

- Terminate the line before adding a new keyword like `then` (you will see `if` and `then` later):

  Bad:

  ```
  if [ 1 -lt 2 ] then
  ```

  Good:

  ```
  if [ 1 -lt 2 ]; then

  if [ 1 -lt 2 ]
  then
  ```

- Quote string variables used in conditions to avoid errors caused by spaces and/or newlines.

  Bad:

  ```
  [ $STR1 = $STR2 ]
  ```

  Good:

  ```
  [ "$STR1" = "$STR2" ]
  ```

3.1.5 Table of conditions

Numeric conditions

Operator	Description
`-eq`	Equal to
`-ne`	Not equal to
`-gt`	Greater than
`-lt`	Less than
`-ge`	Greater than or equal to
`-le`	Less than or equal to

Note that you can use `<` and `>` with double-bracket syntax. For example, `[[ 1 < 2 ]]` or `[[ 5 > 3 ]]`.

File tests conditions

Operator	Description
`-d`	Directory
`-e`	Exists (also -a)
`-f`	Regular file
`-h`	Symbolic link (also -L)

`-r`	Readable by you
`-w`	Writable by you
`-x`	Executable by you
`-s`	Not empty

String-based conditions

Operator	Description
`"str1" = "str2"`	Returns true if the strings are equal. You need to add whitespaces framing the `=`
`"str1" == "str2"`	Similar to `=`. Valid only in double-bracket syntax.
`"str1" != "str2"`	Returns true if the strings are not equal
`"str1" > "str2"`	Returns true if str1 is greater than str2 based on lexicographical order.
`"str1" < "str2"`	Returns true if str1 is smaller than str2 based on lexicographical order.
`-z "string"`	Returns true if the string length is 0.
`-n "string"`	Returns true if the string length is not 0.
`"str1" =~ regex`	Returns true if the specified string matches the extended regex.

3.2 Conditional sentences

In bash, the if/then/else statement is used for conditional execution of commands. Here is the basic syntax:

```
if [ condition1 ]
then
  # commands to be executed if condition1 is true
elif [ condition2 ]
then
  # commands to be executed if condition1 is false and condition2 is true
else
  # commands to be executed if both condition1 and condition2 are false
fi
```

Note that the then keyword must be in a different line. You can use the shell command separator `;` to put the `then` in the same line:

```
if [ condition1 ]; then
  # commands to be executed if condition1 is true
elif [ condition2 ]; then
  # commands to be executed if condition1 is false and condition2 is true
else
  # commands to be executed if both condition1 and condition2 are false
fi
```

3.2.1 One-line conditionals (&& and ||)

In bash, you can use `&&` and `||` as shorthand for writing one-line conditionals.

&& Operator

`&&` is the logical AND operator, which means that the command following `&&` will only be executed if the command preceding `&&` exits with a status code of `0` (success):

```
command1 && command2
```

In this case, if `command1` is successful (exits with a status code of `0`), `command2` will be executed. If `command1` fails (exits with a nonzero status code), `command2` will not be executed.

For example, you could:

```
! [ -d directory_1 ] && mkdir directory_1
```

It will check if the directory exists. If not, the directory will be created by executing the second instruction. You can use whatever command, not only single or double brackets syntax:

```
apt update && apt upgrade
```

This will upgrade the system only if the update command is successful.

|| Operator

`||` is the logical OR operator, which means that the command following `||` will only be executed if the command preceding `||` exits with a status code of nonzero (failure):

```
command1 || command2
```

In this case, if `command1` fails (exits with a nonzero status code), `command2` will be executed. If command1 is successful (exits with a status code of `0`), `command2` will not be executed.

The same example as above could be written using ||:

```
[ -d directory_1 ] || mkdir directory_1
```

You can also use whatever command with ||:

```
which git || sudo apt-get install git
```

This will test if `git` is installed, and install it if not.

You can combine the two operators. The `&&` operator has higher precedence than the `||` operator, which means that `&&` is evaluated first. For example:

```
grep "pattern" myfile.txt && echo "pattern found" || echo "pattern not
found"
```

In this case, if `grep "pattern" myfile.txt` successfully finds the pattern in `myfile.txt`, the message `pattern found` will be printed to the console. If not, the message `pattern not found` will be printed instead.

3.2.2 Case statement

The `case` statement is a control structure in bash that allows you to test a variable against a set of patterns. It's often used to implement menu-like interfaces where the user selects an option from a list of choices.

Here's the basic syntax of the case statement:

```
case variable in
  pattern1)
    commands1
    ;;
  pattern2)
    commands2
    ;;
...
  patternN)
    commandsN
    ;;
  *)
    default-commands
    ;;
esac
```

In this syntax, `variable` is the variable that you want to test, and `pattern1`, `pattern2`, and so on, are the patterns that you want to match against. Each pattern is followed by a block of commands that will be executed if the pattern matches the variable. The `;;` operator is used to mark the end of each block of commands.

You can use bash patterns like:

- `*`: Will match everything
- `option1|option2`: Will match if variable's value is either option1 or option2
- `*.txt`: Will match texts ending in `.txt`
- `file??.txt`: will match `file01.txt`, `file02.txt`, etc.

The `case` statement tries to match variable against the patterns in order, from top to bottom. If a match is found, the commands for that pattern are executed and the case statement terminates. If no match is found, the default-commands block is executed, if it is present.

4. Supplementary material

Shell scripting:

https://bash.cyberciti.biz/guide/Main_Page

Command line evaluation:

https://docstore.mik.ua/orelly/unix/upt/ch08_05.htm

https://www.oreilly.com/library/view/learning-the-bash/1565923472/ch07s03.html

Types of variables:

https://tldp.org/LDP/abs/html/untyped.html

Conditions:

https://acloudguru.com/blog/engineering/conditions-in-bash-scripting-if-statements

Comparison cheat sheet:

https://kapeli.com/cheat_sheets/Bash_Test_Operators.docset/Contents/Resources/Documents/index

Single and double-bracket syntax

https://acloudguru.com/blog/engineering/conditions-in-bash-scripting-if-statements#h-1-single-bracket-syntax

https://developer.ibm.com/tutorials/l-bash-test/