UNIVERSITÉ DU LUXEMBOURG

Operating Systems 1

Christian Grévisse (christian.grevisse@uni.lu)
Saharnaz Dilmaghani (saharnaz.dilmaghani@uni.lu)

Introduction to Bash Scripts

2018-12-11

Abstract

Shell scripts are executable files containing a sequence of commands which can be run in an interpreter. Such scripts can be written in different scripting languages, such as Python, PHP or even Swift. In this document[1], we will present basic aspects of bash scripts, i.e. shell scripts for the bash shell.

1 General Structure of a Script

```
<shebang line>

<command 1>
<command 2>

...
```

On Unix-like systems, such as Linux, different kinds of script languages and environments for executing scripts exist. Bash scripts are one type of such scripts, which are executed by the bash shell. In order to allow the bash shell to determine which environment shall execute the commands in a given script file, every such file contains a special first line, the so-called **Shebang line**. It indicates the path to the environment that shall execute the following commands.

In case of bash scripts the bash shell will execute these commands itself. The first line of a bash script is:

```
#!/bin/bash
```

The commands of a bash script can be any of the commands that can also be executed directly on the command line.

Example 1 (Hello, World!)
This is a very simple script, which just writes *Hello, World!* to standard output:

```
#!/bin/bash
echo "Hello, World!"
```

To run the script, you need to:

- write the above text to a file named, e.g., `helloWorld.sh` (the `.sh` extension is often used for bash scripts)
- give execution rights (`x`) to the corresponding file (verify with `ls -l`, change with `chmod`)
- now the script can be run with
  `$ ./helloWorld.sh`

Output:

```
Hello, World!
```

[1] Based on material by Dr. Laurent Kirsch

2 Bash Script Essentials

2.1 Comments

Comments can be added to a script using the # sign. All text that follows a # sign on the same line is considered to be a comment and is ignored during the script's execution.

2.2 Variables

2.2.1 Variable Names, Assignment & Access

In bash scripts all variables are untyped, i.e. there is no explicit type declaration. Variables are, however, assigned and accessed in different ways.

To *assign* a simple constant value to a variable, the = operator can be used:

```
1 <varname>=<value>
```

> ✎ **Important**
> Do *not* insert any spaces between the variable name, the = operator and the value.

To *access* the current value of a variable, the name of the variable must be preceded by a $ sign.

```
1 ${<numericVariable>} # or $<numericVariable>
2 "${<stringVariable>}" # or "$<stringVariable>"
```

Note that it is not strictly necessary to use " " with string variables. However, it is generally a good idea to use quotes, as otherwise problems arise as soon as these variables store strings containing spaces.

Example 2 (Variable assignment & access)

```
1 #!/bin/bash
2
3 greeting="Hello, World!"
4 echo "$greeting"
5 n=42
6 echo $n
```

```
$ ./variables.sh
Hello, World!
42
```

2.2.2 Variable Manipulation

Strings can be simply reassigned using the = operator. Strings can be concatenated as follows:

```
1 hello="Hello, "
2 world="World!"
3
4 echo "$hello$world"
```

The last line could also be written as

```
1 echo "${hello}World!"
```

The curly braces ({ }) are in this case needed to determine where the first variable name ends.

While strings surrounded by double quotes allow for variable expansion, single quotes prevent it:

```
$ name="Christian"
$ echo "My name is $name"
My name is Christian
$ echo 'My name is $name'
My name is $name
```

Numerical variables can also be reassigned with the = operator. Simple constants can be assigned straight away, whereas the result of an arithmetic expression needs to be included in double parenthesis or preceded by the keyword `let`:

```
1  #!/bin/bash
2
3  n=42
4
5  ((m = 2 * n + 1))
6
7  # Alternative:
8  let m=2*$n+1
9
10 echo "n = $n, m = $m"
```

Note that, in the case of the double parenthesis syntax, it *is* allowed to insert spaces between operators and operands when the entire expression. The operators that can be used in arithmetic expressions are the same as in Java (+, -, *, /, %, ++, --)

2.2.3 Scope of a variable

To make a variable visible outside of a shell, you have to *export* it to the bash environment with the **export** command.

By setting environment variables a command is using, you may alter its functioning. For instance, the **man** command defines, among others, the **PAGER** environment variable. It determines how the text of manual pages is shown.

By executing

```
$ export PAGER=cat
```

the whole manual of a command is shown all at once, whereas by executing

```
$ export PAGER=less
```

the manual is shown page by page, which is the default configuration.

2.2.4 $PATH variable

The **$PATH** variable is an environment variable that stores a colon-separated list of directories the shell looks into in order to execute commands. If the variable contains `/bin/:/usr/bin:/usr/local/bin`, then when executing **cat**, the shell will search for `/bin/cat`, `/usr/bin/cat`, `/usr/local/bin/cat`. If the variable contains **.** (i.e. the current directory), it will also search in the current directory, which enables to execute **script.sh** instead of **./script.sh**.

Again, to change the **$PATH** variable, like any other environment variable, use **export**. It is a good idea to make a backup of the initial content of the variable when altering it, to avoid issues caused by a bad setting.

2.3 Standard Input & Output

As already seen, the **echo** command can be used to print a string to standard output. The **-n** option allows to omit the trailing newline character (**\n**). The **-e** option correctly displays escaped characters, such as newline or tab (**\t**), which would otherwise be treated as regular characters.

The **read** command allows to read user input from **stdin** and store it in a variable.

Example 3 (I/O)

```
1 #!/bin/bash
2 echo "Hi!"
3 echo -n "Please enter your age: "
4 read age
5 ((joke = age - 1))
6 echo -e "Wow!\nYou really have $age years?\tYou look no older than $joke!"
```

```
Hi!
Please enter your age: 26
Wow!
You have 26 years?  You look no older than 25!
```

3 Control Structures

3.1 Conditions

The general structure of conditions is:

```
1 if <condition>; then
2   <list of commands>
3 elif <condition 2>; then
4   <list of commands>
5 else
6   <list of commands>
7 fi
```

There can be several `elif` (`else if` in Java) blocks. `elif` and `else` blocks are optional. The `fi` statement is always added behind the last block (in a new line).

Two valid syntaxes for conditions are described hereafter.

3.1.1 Single-Bracket Syntax

```
1 if␣[␣expression␣];␣then
2 ...
3 fi
```

The structure of `expression` depends on whether the condition is file-based, string-based or arithmetic. Note that it is *very important* to insert a space between `[` and the start of the expression, as well as the end of the expression and `]`.

File-based conditions In all of the following examples, `path` is a variable storing the path to some entry in the file system:

- `[ -d "$path" ]` evaluates to 0 (equivalent to true), if the entry is a directory
- `[ -f "$path" ]` evaluates to 0, if the entry is a regular file
- `[ -r "$path" ]` evaluates to 0, if the script has read rights for the entry
- `[ -w "$path" ]` evaluates to 0, if the script has write rights for the entry
- `[ -x "$path" ]` evaluates to 0, if the script has execute rights for the entry

String-based conditions In all of the following examples, `str1` and `str2` are string variables

- `[ "$str1" == "$str2" ]` evaluates to 0, if both strings are equal
- `[ "$str1" != "$str2" ]` evaluates to 0, if strings are different
- `[ "$str1" \> "$str2" ]` evaluates to 0, if `str1` sorts after `str2` lexicographically
- `[ "$str1" \< "$str2" ]` evaluates to 0, if `str1` sorts before `str2` lexicographically
- `[ -z "$str1" ]` evaluates to 0, if `str1` is the empty string ("")
- `[ -n "$str1" ]` evaluates to 0, if `str1` is not the empty string ("")

Arithmetic conditions In the following examples, `num1` and `num2` are numeric variables

- `[ $num1 -eq $num2 ]` evaluates to 0, if `num1` and `num2` are equal
- `[ $num1 -ne $num2 ]` evaluates to 0, if `num1` and `num2` are not equal
- `[ $num1 -ge $num2 ]` evaluates to 0, if `num1` is greater than or equal to `num2`
- `[ $num1 -gt $num2 ]` evaluates to 0, if `num1` is strictly greater than `num2`
- `[ $num1 -le $num2 ]` evaluates to 0, if `num1` is less than or equal to `num2`
- `[ $num1 -lt $num2 ]` evaluates to 0, if `num1` is strictly less than `num2`

All of the above conditions can be combined using `&&` (and) and `||` (or) operators. They can be negated using the `!` operator.

Example 4 (Conditions – Single-Bracket Syntax)

```
1 #!/bin/bash
2
3 echo "Type a value between 0 and 10:"
4 read value
5 if [ $value -le 10 ] && [ $value -ge 0 ]; then
6   echo "Your value is between 0 and 10"
7 else
8   echo "Your value is not between 0 and 10"
9 fi
```

3.1.2 Double Parenthesis Syntax

The double parenthesis syntax can only be used for arithmetic conditions. The syntax is likely more familiar to Java and C-developers, as the usual operators, like ==, <, <=, !=, >, >= can be used for doing comparisons. Furthermore, && (and) and || (or) can be used for combining expressions. With this syntax, variables are not accessed using the $ operator, but by their plain name.

The arithmetic conditions from above would be written:

`(( num1 == num2 ))` evaluates to 0, if num1 and num2 are equal

`(( num1 != num2 ))` evaluates to 0, if num1 and num2 are not equal

`(( num1 >= num2 ))` evaluates to 0, if num1 is greater or equal than num2

`(( num1 > num2 ))` evaluates to 0, if num1 is strictly greater or equal than num2

`(( num1 <= num2 ))` evaluates to 0, if num1 is less or equal than num2

`(( num1 < num2 ))` evaluates to 0, if num1 is strictly less than num2

Example 5 (Conditions – Double Parenthesis Syntax)

```
#!/bin/bash

echo "Type a value between 0 and 10:"
read value
if (( value <= 10 && value >= 0 )); then
  echo "You entered a correct value"
else
  echo "Your value is not between 0 and 10"
fi
```

3.1.3 Case

case is the equivalent of the Java `switch/case` statement. An arbitrary number of cases can be handled. The different cases are checked one by one and as soon as one of them matches the value of `<varname>`, the corresponding list of commands is executed. Subsequent cases are automatically skipped. `*)` is equivalent to Java's `default` case.

```
case "$<varname>" in
  "String1")
    <list of commands>
    ;;
  "String2")
    <list of commands>
    ;;
  *)
    <list of commands>
    ;;
esac
```

A concrete example is given in section 3.2.5.

3.2 Loops

3.2.1 Globbing

Bash supports *globbing*, i.e. the expansion of wildcard characters, often used in the context of filenames.

`*` represents any sequence of characters

```
ls -l *.jpg # displays information about all files with the .jpg extension
ls -l a*.txt # displays information about all text files starting with "a"
```

`?` represents a single character

```
ls -l file?.txt # displays information on file1.txt but not file10.txt (if existing in
    working directory)
```

`[]` matches any character indicated between `[` and `]`

```
ls -l file[A-HXZ] # displays information on fileA, fileB, ..., fileH, fileX and fileZ
```

3.2.2 `for`-loop

The `for`-loop iterates successively over all entries of a given list:

```
1 for <varname> in <array>; do
2   <list of commands>
3 done
```

Example 6 (Basic example)

```
1 #!/bin/bash
2
3 for animal in "fox" "dog" "cat"; do
4   echo "$animal"
5 done
```

```
fox
dog
cat
```

Example 7 (Print the names of all entries in the current working directory)

```
#!/bin/bash

for path in *; do
     echo "$path"
done
```

3.2.3 `while`-loop

The `while`-loop executes the commands in its body as long as the condition remains true. The format of the condition is the same as in `if`-statements.

```
while <condition>; do
  <list of commands>
done
```

Example 8 (Print numbers from 1 to 10)

```
#!/bin/bash

i=1;
while [ $i -le 10 ]; do
  echo -n "$i "
  (( ++i ))
done
```

```
1 2 3 4 5 6 7 8 9 10
```

3.2.4 `until`-loop

The `until`-loop executes the commands in its body as long as the condition remains *false* (i.e., *until* the condition becomes *true*). The format of the condition is the same as in `if`-statements.

```
until <condition>; do
  <list of commands>
done
```

Example 9 (Print numbers from 1 to 10)

```
#!/bin/bash

i=1;
until [ $i -gt 10 ]; do
  echo -n "$i "
  (( ++i ))
done
```

3.2.5 `select`-loop

The `select`-loop allows for creating simple menus

```
select <varname> in <array>; do
  <list of commands>
done
```

`<array>` is an array that contains the choices that the user may choose from. The different options are numbered as they are presented to the user. At each iteration the user makes his choice by typing the number of the option he wants to choose. His choice is stored in `<varname>`.

Example 10 (Simple menu with `select`)

```
#!/bin/bash

select choice in "hello" "quit"; do
  case "$choice" in
    "hello")
      echo "hello"
      ;;
    "quit")
      echo "bye"
      break
      ;;
    *)
      echo "that's not a valid choice"
      ;;
    esac
done
```

As long as the user types 1, the `hello` case is executed. As soon as the user types 2, the `quit` case is executed and the loop is terminated because of the `break` statement. Note that the `break` and `continue` statements known from Java can be used within any of the presented loops and have the same effect as in Java.

4 Commands

4.1 Command line arguments

There are special, pre-defined variables which store the command line parameters:

`$1` stores the first parameter

`$2` stores the second parameter ...

`$#` stores the number of parameters

`$*` is a string consisting of all parameters

`$@` is an array containing all parameters, which is useful for iterating over them

`$0` returns the name of the shell script

The `shift` command allows to shift all parameters to the left, i.e.: `$2` becomes `$1`, `$3` becomes `$2` etc., while `$#` gets decremented. Shifting is another standard technique that is useful for iterating over all command line arguments, especially when there is a variable number of them.

Example 11 (Iterating over command line arguments)

```
1 #!/bin/bash
2
3 for arg in $@; do
4   echo $arg
5 done
```

```
./cla.sh bla bli blubb
bla
bli
blubb
```

PID of a Script
Another pre-defined variable beginning with $ is $$, which returns the PID of the executed script. This might be useful in the context of process management and signaling.

4.2 Command Substitution

Command substitution allows the output of a command to replace the command itself. Command substitution occurs when a command is enclosed as follows:

`$(command)` or `‘command‘`

Bash performs the expansion by executing command and replacing the command substitution with the standard output of the command.

Example 12

Assume a text file `filenames.txt` with the following content:

```
$ cat filenames.txt
a.txt b.txt c.txt d.txt
```

Now, when executing

```
$ rm -i ‘cat filenames.txt‘
```

the inner command `cat filenames.txt` is first evaluated and its output (i.e. the file's content) will replace the command itself, yielding the following command to be executed in the second step:

```
$ rm -i a.txt b.txt c.txt d.txt
```

which will delete these 4 files (if they exist).

Command substitution can also be used to store the output of a command in a variable.

Example 13

```
$ filenames = $(cat filenames.txt)
$ rm -i $filenames
```

yields the same result as the previous command.

4.3 Command Execution

We already saw in previous labs that you can execute several commands in parallel by separating them with `;`:

`$ (<command1> &) ; (<command2> &)`

Commands can also be sequentially chained together with the && and || operators. In a so-called *and-list* `<command1> && <command2> && ... && <commandn>`, each command is executed in turn until one command fails (i.e. has a return value of true, i.e. an exit status of 0). Similarly, in an *or-list* `<command1> || <command2> || ... <commandn>`, each command is executed in turn as long as the previous command fails (i.e. has a non-zero exit status).

5 Preview: Other Scripting Languages

As already mentioned in the abstract, there are plenty more fish in the sea! While the basic *Hello, World!* example in *Bash* is ...

```
#!/bin/bash

world="World!"
echo "Hello, $world"
```

... the corresponding script in *PHP*[2] would be ...

```
#!/usr/bin/php

<?php
  $world = "World!";
  echo "Hello, $world\n";
?>
```

... whereas in *Python*[3] ...

```
#!/usr/bin/python3

if __name__ == "__main__":
  world = "World!"
  print("Hello, {}".format(world))
```

... or, in Apple's pretty recent language *Swift*[4] ...

```
#!/home/student/swift/usr/bin/swift

let world = "World!"
print("Hello, \(world)")
```

Of course, the execution of these scripts requires the respective interpreter to be installed and its location in the `$PATH` variable. You will see more on these languages in other lectures during the next semesters.

[2] http://php.net
[3] https://www.python.org
[4] https://swift.org