Princípios da Computação

Shell scripts

What is a shell script?

- A shell script is a computer program that is interpreted by a shell.

- It combines in a single file a sequence of commands that would otherwise have to be typed on the keyboard one at a time.

- Usually created to automate command sequences that are frequently used, thus saving time.

How is a shell script?

- It is a text file composed of:
    - The interpreter directive (the first line of the script)
    - Commands
    - Environment variables
    - Control-flow constructs (decision and loops)

The interpreter directive

- The very first line of a script is the *interpreter directive*.
- It is a magic comment that tells Unix-based operating systems how to execute your file.
- Starts with the *shebang*: **`#!`**
    - This two-byte combination states this is an executable script file.
- Follows the interpreter file name (with path): **`/bin/bash`**

Why write the interpreter directive

- There are several scripting languages.
    - In this way, each script is executed by the appropriate interpreter.
- bash: **`#!/bin/bash`**

- But there are many others…
    - csh: **`#!/bin/csh`**
    - Python: **`#!/usr/bin/python3`**
    - Perl: **`#!/usr/bin/perl`**
    - Ruby: **`#!/usr/bin/env ruby`**

The "Hello World!" script

- The simplest shell script is a sequence of commands.
- The "Hello world" script can be written like this:

```
#!/bin/bash


echo 'Hello world!'
```

Variables

- You can use variables in your scripts.
    - You can make use of the global environment variables.
        - Normally named in uppercase: e.g. **`HOME`**, **`USER`**.
- You can also create your local variables and use them to store data relevant for your script.
    - Usually named in lowercase: e.g. **`my_var`**, **`install_dir`**.

Assigning a value to a variable

- Shell variables store **text strings**!

- A variable is declared in its first use.

- The equal sign **=** assigns a value to a variable.

    - No spaces are allowed around the **=** symbol!!!

```
hello_msg='Hello, folks!'
other_variable="It's easy to assign a value."
```

Retrieving the value of a variable

- Use the variable expansion symbol: **$**
- The shell will replace the variable name by its value.

```
hello_msg='Hello, folks!'
echo $hello_msg How are you doing?
```

Some useful global variables

HOME	Path to the user home directory.
PWD	Current working directory.
USER	The username of the current user.
$	Process identification number of the current program.
?	Status of the last command executed.

Reading user input from the keyboard

- We use the **`read`** command, followed by a variable that will store the input.
    - Use the -p option to print a prompt.

```
read -p 'Name: ' name
echo "Hi, $name. How are you doing?
```

Embedding a command into an expression

- The output of a command can be captured and embedded into a text string.

- Use the sub shell: **`$(`*`command`*`)`**

```
today="Today is $(date)."
echo $today
```

Command line arguments

- The command line arguments are passed to the script in the form of variables.

```
$ ./script.sh arg1 arg2 arg3
```

$#	Number of arguments.
$*	List of all arguments
$0	Name of the shell script.
$1, $2, …	First, second, … arguments.

Control flow

isep Instituto Superior de Engenharia do Porto

Decision: `if` statement

- Keywords: **`if`** - **`then`** - **`elif`** - **`else`** - **`fi`**

```
if condition1; then
   # Some action here.
elif condition2; then
   # (Optional) Alternative action here.
   # (There may be more elif's)
else
   # (Optional) Last alternative action here.
fi
```

Testing a condition

- Use the **`test`** keyword:
  - **`test`** *`condition`*
- Use the **[ ]** operator:
  - `[ condition ]`
  - Note: spaces around the **[ ]** operator are mandatory, otherwise syntax error!

Testing a condition

```
if test $colour = "BLACK"; then
   echo "Black"
elif [ $colour = "WHITE" ]; then
   echo "White"
else
   echo "Some colour"
fi
```

Testing strings

`"$string"`	TRUE if **`string`** is not empty.
`"$str1" = "$str2"`	TRUE if strings are equal.
`"$str1" != "$str2"`	TRUE if strings are different.
`-n "$string"`	TRUE if **`string`** is not empty.
`-z "$string"`	TRUE if **`string`** is empty.

We enclose variables inside double quotes to prevent a syntax error, in case a variable is empty!

Testing integers

"$val1" -eq "$val2"	TRUE values are equal.
"$val1" -ne "$val2"	TRUE if values are different.
"$val1" -gt "$val2"	TRUE if **val1** is greater than **val2**.
"$val1" -ge "$val2"	TRUE if **val1** is greater than or equal to **val2**.
"$val1" -lt "$val2"	TRUE if **val1** is less than **val2**.
"$val1" -le "$val2"	TRUE if **val1** is less than or equal to **val2**.

We enclose variables inside double quotes to prevent a syntax error, in case a variable is empty!

Testing the file system

-f "$name"	TRUE if $name is an existing file.
-d "$name"	TRUE if $name is an existing directory.
-r "$name"	TRUE if $name has read permission.
-w "$name"	TRUE if $name has write permission.
-x "$name"	TRUE if $name has execute permission.
-s "$name"	TRUE if size of file $name is not zero.

We enclose variables inside double quotes to prevent a syntax error, in case a variable is empty!

Testing multiple conditions

- AND: `[ condition1 ] && [ condition2 ] …`
- OR: `[ condition1 ] || [ condition2 ] …`

```
if [ $? = 0 ] && [ -f "$file" ]; then
  echo "Command successful."
  echo "Result written to $file."
else
  echo "Houston, we have a problem."
fi
```

Loop: `for` statement

- Keywords: **`for`** - **`in`** - **`do`** - **`done`**

- A variable successively takes the values from a list.

```
for variable in list of values; do
   # Some action here.
done
```

Loop: `for` statement

```
for name in Huey Dewey Louie; do
  echo "Hey, $name\!"
done
```

Output:

```
Hey, Huey!
Hey, Dewey!
Hey, Louie!
```

Frequently used lists

The command line arguments.

```
for arg; do
  echo "This is a command line argument: $arg"
done
```

The names of all entries in the current working directory.

```
for entry in *; do
  echo "File/directory: $entry"
done
```