Lab 5

Shell programming
Shell scripting

The shell can also be used to interpret the contents of files containing commands and to execute them. These command files are called *scripts*. They can be seen as executable programs in text format. For this, the script file, *my-first-script* in the example below, must have execute permissions set at least for the owner of the file:

```
$ chmod u+x my-first-script
```

After the execute permissions have been added, the script file can be executed like any compiled executable file; for example, considering the file *my-first-script* above:

```
$ ./my-first-script
```

The first line of the script file has a special syntax that specifies the program that will interpret the script. This first line instructs the shell which command interpreter to launch in order to interpret and execute the commands in the script. For example, a *bash* shell can launch the script file above, *my-first-script*, which has been given execute rights as above and which contains the following commands:

```
#!/bin/bash

# This is an ordinary comment

echo "Hello scripting world !"
exit 0
```

The first line is a special comment (it starts with the character #, immediately followed by !) which specifies the command interpreter (*/bin/bash*) used to interpret and execute the commands that follow. The last line, which calls

the command *exit* with the parameter 0, illustrates a good practice in Unix programming in general (and shell scripting is no exception), whereby the termination code of the program (of the script, in the case above) is communicated to the parent process (in our case, the shell that launched the script above). The Unix convention says that any program that ends without error returns the code 0. Any other return value represents, a priori, an erroneous termination of the program.

Alternatively, the script *my-first-script* can be run without setting execute permissions for users, by invoking the appropriate interpreter (`/bin/bash` in our example) as below:

```
$ bash my-first-script
Hello scripting world !
$ echo $?
0
$
```

In the most general case, the first line of the script can designate any type of interpreter, which will of course be used to interpret the commands that follow in the script. Here is an *awk* script, which can also be launched by any other command interpreter, including `bash`, and which prints the arguments received by the script on the command line:

```
#!/usr/bin/awk -f

BEGIN   {
        for(i = 0; i < ARGC; i++)
                printf "ARGV[%d] = %s\n", i, ARGV[i]
        exit
}
```

If the script above is saved in a file named `print-args.sh`, it can be executed in the following ways:

```
$ awk -f print-args.sh 1 2 3
ARGV[0] = awk
ARGV[1] = 1
ARGV[2] = 2
ARGV[3] = 3
$ chmod u+x print-args.sh
$ ./print-args.sh 1 2 3
ARGV[0] = awk
ARGV[1] = 1
ARGV[2] = 2
ARGV[3] = 3
```

As can be seen, *awk* is practically a programming language with a structure close to that of compiled programming languages.

In fact, all command interpreters also have their own programming language, with their own syntax and their own commands. In what follows, the main "statements" understood by the Bourne Again Shell will be presented: loops (while/for), conditional statements (if/case) and functions.

1 Test

As we saw in the previous lab, the test operation commonly used in conditional or iterative statements is implemented by *bash* with the help of the built-in command *test* or *[*. In this lab, we will extend the examples of using the command *test* so that it can also be used for files, after having seen in the previous lab how it can be used to test strings and numbers. If needed, review the examples from the previous lab which, together with the examples that follow, will help us understand how the command *test* can be used together with conditional and iterative statements.

```
$ X=somefile
$ [ -f "$X" ] && \
>       echo "X is the path of a real file" || \
>       echo "No such file: $X"
No such file: somefile

$ X=/etc/passwd
$ [ -f "$X" ] && \
>       echo "X is the path of a real file" || \
>       echo "No such file: $X"
X is the path of a real file

$ X=/bin/ls
$ [ -x "$X" ] && \
>       echo "X is the path of an executable file" || \
>       echo "X is not the path of an executable file"
X is the path of an executable file
$ Y=/usr/bin/wc
$ [ -x "$X" -a -x "$Y" ] && \
>       $(echo $X -l) | $(echo $Y -l) || \
>       echo "Don't know how to count output lines"
25
$ Y=./my-word-count
$ [ -x "$X" -a -x "$Y" ] && \
>       $(echo $X -l) | $(echo $Y -l) || \
>       echo "Don't know how to count output lines"
Don't know how to count output lines
$ $ cat > my-word-count
#!/bin/sh
wc -l
```

```
Ctrl-d
$ chmod u+x my-word-count
$ [ -x "$X" -a -x "$Y" ] && \
>       $(echo $X -l) | $(echo $Y -l) || \
>       echo "Don't know how to count output lines"
26
$

$ X=.
$ [ "$X" -nt "/etc/passwd" ] && \
>       echo "$X is a file which is newer than /etc/passwd"
. is a file which is newer than /etc/passwd
$ [ -d "$X" -o -f "$X" ] && [ "$X" -ot "/etc/passwd" ] && \
>       echo "$X is a `[ -f "$X" ] && echo file || echo directory` \
>                       older than /etc/passwd" || \
>       [ ! -e "$X" ] && echo "$X not found" || \
>       echo "$X is a `[ -f "$X" ] && echo file || echo directory` \
>                       newer than /etc/passwd"
. is a directory newer than /etc/passwd
$ X=my-word-count
$ [ -d "$X" -o -f "$X" ] && [ "$X" -ot "/etc/passwd" ] && \
>       echo "$X is a `[ -f "$X" ] && echo file || echo directory` \
>                       older than /etc/passwd" || \
>       [ ! -e "$X" ] && echo "$X not found" || \
>       echo "$X is a `[ -f "$X" ] && echo file || echo directory` \
>                       newer than /etc/passwd"
my-word-count is a file newer than /etc/passwd
$ X=somefile
$ [ -d "$X" -o -f "$X" ] && [ "$X" -ot "/etc/passwd" ] && \
>       echo "$X is a `[ -f "$X" ] && echo file || echo directory` \
>                       older than /etc/passwd" || \
>       [ ! -e "$X" ] && echo "$X not found" || \
>       echo "$X is a `[ -f "$X" ] && echo file || echo directory` \
>                       newer than /etc/passwd"
somefile not found
$
```

This last command tests a simple piece of functionality that tries to capture whether changes to files or directories in the current directory (located somewhere in the subtree rooted at $HOME, i.e. the home directory) are later than the last modification of the file `/etc/passwd` (potentially even of the entry corresponding to the user). Unfortunately, expressing this relatively simple functionality in a single command, and in the absence of high-level conditional constructs (of the programming language type), makes the task relatively complicated and rather error-prone (either syntax or logic errors). This is a good reason, among many others, to highlight in what follows the usefulness of scripts

and of the programming language constructs supported by the shell.

2 If and Case statements

Conditional *if* statements in the shell look as follows:

```
if [ ... ]
then
  # if-code
else
  # else-code
fi
```

Notice the position of the keywords *then*, *else* and *fi*, respectively. They must appear alone on a line, right at its beginning. To save space, the character ";" can be used, which allows two consecutive lines to be joined on the same line:

```
if [ ... ]; then
  # if-code
else
  # else-code
fi
```

Several *if* statements can be nested using the statement *elif*, as in the following script, which you can save in the file *if.sh* (for this script, as well as for all the ones that follow, do not forget to set the execute permission before running the script):

```
#!/bin/bash

read filename
if  [ ! -e "$filename" ]; then
 echo "$filename does not exist"
 elif [ -f "$filename" ]; then
   echo "$filename is a regular file"
 else
   echo "$filename exists, but we don't know what kind of file it is"
fi
```

Running the script highlights the functionality of the *if* statement:

```
$ ./if.sh
somefile
somefile does not exist
$ ./if.sh
/etc/passwd
/etc/passwd is a regular file
$ ./if.sh
/etc
```

```
/etc exists, but we don't know what kind of file it is
$
```

To make the script above somewhat more complete, the *case* statement can be used, which has the following syntax:

```
case var in
        pattern1)
                cmds
                ;;
        pattern2)
                cmds
                ;;
        *)
                defaultcmd
esac
```

The effect of the command is to match the contents of the variable `var` against one of the specified patterns and, as a result of the match, to execute the commands corresponding to the matched pattern. If no pattern matches, the default command specified for the wildcard variant * is executed. Under these conditions, the script that tests the type of a file whose name is read from the keyboard looks as below (we will call the corresponding file *case.sh*):

```
#!/bin/bash

echo -n "Please input a filename: "
read filename
longformat=`ls -ld $filename 2> /dev/null`
case "${longformat:0:1}" in
        -)
                echo "$filename is a regular file"
                ;;
        d)
                echo "$filename is a directory"
                ;;
        b)
                echo "$filename is a block file"
                ;;
        c)
                echo "$filename is a character file"
                ;;
        *)
                echo "Sorry, I don't know anything about $filename"
                ;;
esac
```

Recall from the previous labs that any command that appears between back-quotes is replaced by a string representing the result of executing the command

between the backquotes. The redirection of *stderr*, represented by file descriptor 2, is necessary to handle the situation in which the name of the file typed on the keyboard does not correspond to an existing file. */dev/null* is a character file with a special behaviour in the system: any write to */dev/null* is lost, and nothing can be read from */dev/null*. Consequently, redirecting *stderr* to */dev/null* has the effect of discarding the error message generated by the command *ls -l* when the file name read from the keyboard does not correspond to an existing file in the system.

As mentioned in the lecture, listing the attributes of a file in long format has the effect, among other things, of printing the file type on the screen as the first character. To isolate this character within the string generated by the command *ls -l*, the syntax *${variable_name:offset:length}* is used, which makes it possible to obtain a substring of the variable *variable_name* that starts at position *offset* and has length *length*. Concretely, the contents of the line displayed by the command *ls -ld* are stored in the variable `longformat`, and the expression *${longformat:0:1}* will extract 1 character located at offset 0 within the string represented by the variable *longformat*; in other words, the first character of the string.

The results of running the script above look as follows:

```
$ ./case.sh
Please input a filename: /etc
/etc is a directory
$ ./case.sh
Please input a filename: /etc/passwd
/etc/passwd is a regular file
$ ./case.sh
Please input a filename: /dev/tty1
/dev/tty1 is a character file
$ ./case.sh
Please input a filename: /dev/sda1
/dev/sda1 is a block file
$ ./case.sh
Please input a filename: somefile
Sorry, I don't know anything about somefile
$
```

3 Iterative statements

Interpreted languages also offer functionality in the form of iterative statements, i.e. statements executed repeatedly in a loop. The main syntactic constructs offered by bash for implementing loops are the *for* and *while* statements.

3.1 For loops

for statements iterate through a list of values. Execute the following script, which you save in a file *myfor-loop.sh*:

```
$ cat -> myfor-loop.sh
#!/bin/bash
for i in 1 2 3 4 5
do
  echo "Iteration number $i"
done
$ chmod u+x myfor-loop.sh
$ ./myfor-loop.sh
Iteration number 1
Iteration number 2
Iteration number 3
Iteration number 4
Iteration number 5
$
```

Notice that *do* and *done*, keywords of the *for* statement, must appear on a separate line, right at its beginning. If you modify the script and move the keyword *do* onto the same line as *for*, you will get a syntax error when you execute the script. You can fix this error using the special character *;*, which joins two lines:

```
#!/bin/bash
for i in 1 2 3 4 5; do
  echo "Iteration number $i"
done
```

The values in the list can be practically anything. For instance, try to execute the following script, which you can save in the file *myfor-loop2.sh*:

```
#!/bin/bash
for i in math 1.5 * 2 done
do
  echo "The value of i is $i"
done
```

To understand the behaviour of the *for* statement in this case, it is important to recall the interpretation of wildcards in bash presented in the lecture. Try running the script above with and without *. And what if you use the escape character * ? And if you use *, what happens if, in the body of the *for* statement, you display the value of the variable *i* without enclosing it in quotation marks (as below)?

```
echo The value of i is $i
```

3.2 While loops

The *while* statement is most often used together with the command *test*, as in the example below, which you can save in the file *while.sh*:

```
#!/bin/bash
while [ "${INPUT_STRING:-hello}" != "bye" ]
do
  echo "Enter some data (bye to exit the loop)"
  read INPUT_STRING
  echo "You entered: $INPUT_STRING"
done
```

The script above reads character strings from the keyboard in a loop until "bye" is typed. What happens if the variable INPUT_STRING has no initial value?

The script above can be modified into the following variant (*while2.sh*), which does not use the test command but the character ":" (equivalent to the command *true*), which always returns the value true:

```
#!/bin/bash
while :
do
  echo "Enter some data (Ctrl-C to exit the loop)"
  read INPUT_STRING
  echo "You entered: $INPUT_STRING"
done
```

Also, the *while* statement is often used together with *read*. As an example, you have the script below (*while3.sh*), which modifies the previous script *case.sh* to allow file names to be entered repeatedly:

```
#!/bin/bash

echo -n "Please input a filename: "
while read filename
do
        longformat=`ls -ld $filename 2> /dev/null`
        case "${longformat:0:1}" in
                -)
                        echo "$filename is a regular file"
                        ;;
                d)
                        echo "$filename is a directory"
                        ;;
                b)
                        echo "$filename is a block file"
                        ;;
                c)
```

```
                        echo "$filename is a character file"
                        ;;
                *)
                        echo "Sorry, I don't know anything about $filename"
                        ;;
        esac
        echo -n "Please input a filename: "
done
```

You can end the data input with Ctrl-d or terminate the program with Ctrl-c.

4 More about variables

Besides the shell variables (internal variables and environment variables), there is a set of special variables which, most of the time, cannot be modified.

The first set of variables of interest is that of the variables *0, 1, ... 9*. The variable *$0*, for instance, represents the name of the program, generally called the *basename*, while the rest of the variables up to 9, *$1, $2,..., $9*, represent the parameters with which the script was called. The variable @ represents these parameters, whose exact number is stored in the variable #. For a better understanding of these aspects, run the following script, *param.sh*:

```
#!/bin/sh

echo "I was called with $# parameters"
echo "My name is $0"
echo "My nicer name is `basename $0`"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

The external command *basename* is used to remove the path from the name of the script. Here are the results of a possible run:

```
$ ./param.sh
I was called with 0 parameters
My name is ./param.sh
My nicer name is param.sh
My first parameter is
My second parameter is
All parameters are
$ ./param.sh first second third fourth
I was called with 4 parameters
My name is ./param.sh
My nicer name is param.sh
My first parameter is first
```

```
My second parameter is second
All parameters are first second third fourth
$
```

With the help of the command *shift*, more than 9 parameters can be used on the command line. The following script, *param2.sh*, highlights the function of the command *shift*, which, when called, iterates through the call parameters of the script ("shifts" the parameters to the left). As the parameters are shifted, their number, contained in the variable $#, decreases.

```
#!/bin/sh
while [ "$#" -gt "0" ]
do
  echo "\$1 (first parameter) is $1"
  shift
done
```

The result of executing the command demonstrates how all the call parameters can be accessed, regardless of their number:

```
$ ./param2.sh 1 2 3 4
$1 (first parameter) is 1
$1 (first parameter) is 2
$1 (first parameter) is 3
$1 (first parameter) is 4
$
```

Finally, as discussed in the lecture and in the previous labs, the variable *?* contains the value of the return code with which the previous command ended. The value of this return code can be used to notify the user about the result of executing a command, as in the script below (*retur.sh*):

```
#!/bin/sh

read cmd
eval "$cmd" 2> /dev/null
if [ "$?" -ne "0" ]; then
  echo "$cmd has failed!"
fi
```

The built-in shell command *eval* evaluates the string given as an argument as a command and returns the return code of this command. Here are some examples of running it:

```
$ ./retur.sh
somecmd
somecmd has failed!
$ ./retur.sh
[ "0" -gt "1" ]
[ "0" -gt "1" ] has failed!
$
```

5 Lab assignments

1. Do as many of the exercises presented in this lab as possible. Use a text editor to create the scripts presented in the lab more quickly and to test them. Do not copy the code from the pdf; you may introduce invisible special characters from the pdf file which prevent the execution of script files created by copy-pasting from the pdf.

2. Modify the script `case.sh` so that it receives the file name as a parameter on the command line, instead of reading it from the keyboard. Then add code so that the script can process several files given as parameters on the command line, for example as below:

   ```
   $ ./case.sh /etc/passwd . `tty`/dev/sda1
   ```

3. Write a script that implements your own version of the command *ls* without any parameter (i.e., to list the contents of the current directory). Then modify the script to list the contents of a directory given as a call parameter of the script.

4. Write a script that uses a pipeline command to display the users and the PIDs of their processes, as displayed by the command *ps auxw*. The script must capture in a pipe the output of the command *ps auxw* and iterate through each line, printing the first two fields of the output, those corresponding to the user and the PID. Pass a user name as a parameter on the command line and modify the script to display how many processes that user has running.

5. Write a script *myfind* that emulates the simplified behaviour of the command *find* with the flags *-name*, *-type* and *-exec*. The command receives a directory as its first parameter and does not work recursively (i.e., maxdepth = 1). It must be able to find a file by name and/or type and, once found, be able to execute a command on it. The name, the type and the command to execute are given as parameters on the command line, exactly as for the command *find (man find)*.