Sistemas de Operação / Operating Systems

(2018/2019 academic year)

Lesson script

Script #01 — The bash scripting language

Summary

Shell scripting using bash.

Exercise 1 *Familiarization with common Unix/Linux commands and internal* `bash` *commands.*

1. *Command* `man` *shows a description page for the command passed as argument. Use it to see the role of the commands:* `ls`, `mkdir`, `rmdir`, `pwd`, `rm`, `mv`, `cat`, `echo`, `less`, `head`, `tail`, `cp`, `diff`, `wc`, `sort`, `grep`, `sed`, `tr`, `cut`, `paste`, `chmod`, `stat`, `history`, ...

2. *Commands usually have switches/options that allow you to change their default behavior. Explore some of the* `ls` *command options:* `-l`, `-a`, `-lh`, `-R` ...

3. *In general, commands correspond to executable programs, but there are some that are internal functions of the command interpreter. Command* `cd` *is one of them. To see a list of such commands, execute the internal command* `help`. *To see a description of a given internal command, execute* `help` *followed by its name, as, for instance,* `help cd`.

Exercise 2 *Redirecting input and output.*

Many commands receive input from the standard input *(usually the keyboard) and deliver output to the* standard output *and the* standard error *(usually the display monitor in both cases). However, it is possible to change this default behaviour.*

1. *Operators* `>` *and* `>>` *redirect the* standard output *to a given file. The following sequence of commands illustrates their use.*

```
echo ola
echo ola > z
cat z
echo hello >> z
cat z
echo hola > z
cat z
```

2. *Operator* `2>` *redirects the* standard error *to a given file. The following sequence of commands illustrates its use.*

```
rm -f zzz          # to guarantee zzz does not exist
cat zzz            # an error occurred and an error message appears
cat zzz > out      # the error message was not redirected. Why?
cat out            # file zzz it is empty. Why?
cat zzz 2> err     # the error message was redirected. Why?
cat err
```

3. *The following exercice illustrates redirection of* standard output *and* standard error, *using terminals. (Terminals are seen as files from the operating system point of view.)*

   - *Open 3 different terminals. (Usually the sequence of keys CRTL+ALT+t opens a terminal.)*

- *In each one of the terminal run command* `tty` *and keep the results, as they are the names of the files that represent the terminals. (The names should be be something like* `/dev/pts/N`*, where* `N` *is a number.) Let consider their names as* `/dev/pts/A`*,* `/dev/pts/B` *and* `/dev/pts/C`*.*
- *In terminal* `/dev/pts/A`*, execute comand* `cat /etc/passwd`*. The contents of file /etc/passwd is displayed on the same terminal.*
- *In the same terminal, execute command* `cat/etc/passwd > zzz`*. Now the information in sent to file* `zzz`*. You can check that executing command* `cat zzz`*.*
- *In the same terminal, execute command* `cat /etc/passwd > /dev/pts/B 2> /dev/pts/C`*. Where is the information displayed?*
- *In the same terminal, consider you misspell the name of the file and execute, instead,* `cat /etc/psswd > /dev/pts/B 2> /dev/pts/C`*. Where is the error message displayed?*
- *It should be clear now the meaning of* `>` *and* `2>` *based of the destination of the file contents and the error message. It not, ask for help.*

4. *Operators* `2>&1` *and* `>&2` *(or* `1>&2`*) redirect the* standard error *to the* standard output *and the* standard output *to the* standard error*, respectively. The following sequence of commands illustrates their use.*

```
rm -f zzz          # to guarantee zzz does not exist
cat zzz > err
cat zzz > err 2>&1
cat err
cat /etc/passwd 2> z
cat /etc/passwd 2> z >&2
```

5. *Operator* `|` *redirects the* standard output *of a command to the* standard input *of the next. The following sequence of commands illustrates its use. (Execute* `man wc`*, if you do not know the role of command* `wc`*.)*

```
cat /etc/passwd
cat /etc/passwd | wc -l
```

Exercise 3 *Using special characters.*

1. *Analysing the result of the execution of the following sequence of commands, try to understand the meaning of characters* `*` *and* `?`

```
mkdir dir1
cd dir1
touch a a1 a2 a3 a11 b b1 b11
ls
ls a*
ls a?
ls *
```

2. *Analysing the result of the execution of the following sequence of commands, try to understand the meaning of characters* `[` *and* `]`

```
touch a a1 a2 a3 a11 b b1 b11 c c11
ls
ls [ac]
ls [a-c]
ls [a-c]?
ls [ab]*
```

3. *Character* `\\` *can be used to disable the special meaning of the character following it. The following sequence of commands illustrates its use.*

```
touch a1 a2 a3 a4 a22   # create some files
echo a*
echo a\*
echo a?
echo a\?
echo a\[
echo a\\
```

4. *Characters* `'` *and* `"` *can be used to disable the special meaning of a sequence of characters. The following sequence of commands illustrates their use.*

```
touch a1 a2 a3 a4 a22   # create some files
echo a*
echo "a*"
echo 'a*'
```

Exercise 4 *Declaring and using variables.*

1. *Variables are supported in* `bash`*. The following sequence of commands illustrates their use.*

```
x=abc
xx=0123456789
echo $x
echo $xx
echo ${xx}
echo ${x}x
touch a1 a2 a3 a4 a22   # create some files
z=a*
ls $z
```

2. *In the previous exercise, characters* `"` *e* `'` *seem to be equivalent. Try to understand how they differ executing and analysing the following sequence of commands.*

```
v=a*                # a* is not expanded in assignment
echo $v
echo "$v"
echo '$v'
```

3. *There are functions to manipulate the value of variables. The following sequence of commands illustrates the use of the substring and substitute functions.*

```
x=0123456789
echo ${x:2:4}
echo ${x/123/ccc}
```

4. *Use the manpage of* `bash` *(*`man bash`*) to see other possible manipulations of variables.*

Exercise 5 *Declaring and using functions.*

1. *Functions are supported in* `bash`. *The following sequence of commands illustrates the declaration and use of a function.*

```
# the following code declares function x
x()
{
    ls -l
}
# the following code uses the function x previously defined
x
x | wc -l
```

2. *Functions can accept arguments. Variables* `$1`, `$2`, ..., `$*`, `$@` *and* `$#` *can be used to access them.*

```
y()
{
    echo $#      # the number of arguments
    echo $1      # the first argument
    echo $2      # the second argument
    echo $*      # the list of all arguments
    echo $@      # idem
    echo "$*"    # idem
    echo "$@"    # idem
}
y a bb ccc dddd eeeee
y a "b  b" ccc "dd    dd" eeeee
```

Exercise 6 *Grouping commands.*

1. *Characters* `{ e }` *can be used to group commands. In the following example, the output of a group of commands is redirected to a file.*

```
{
    ls
    echo ========================
    ls
} > z
cat z
```

2. *Characters* `( e )` *can also be used to group commands. In the following example, the output of a group of commands is redirected to a file.*

```
(
    ls
    echo ========================
    ls
) > z
cat z
```

3. *The difference berween them is that in the second case the execution happens in a new instance of the bash. The following bash code shows the differences. Pay attention to the successive values of variable* `zzz`.

```
zzz=abc
echo $zzz
{
    echo "This takes place in the same bash instance"
    zzz=xpto
    echo $zzz
}
echo $zzz
echo ============================
zzz=abc
echo $zzz
(
    echo "This takes place in a new bash instance"
    zzz=xpto
    echo $zzz
)
echo $zzz           # the assignment within !()! 'got lost'
```

Exercise 7 *The conditional* `if` *construction.*

1. *Commands have a return value, that, in C/C++ programs, corresponds to the argument of the* `return` *instruction of the* `main` *function or to the argument of the* `exit` *function. The* `bash` *saves this return value in variable* `$?`*. Execute the following sequence of commands to see it.*

```
ls
echo $?
rm -f zzz     # to guarantee file zzz does not exist
echo $?
test -f zzz
echo $?
touch zzz     # to guarantee file zzz exists
test -f zzz
echo $?
```

2. *The value of* `$?` *is used by* `bash` *as a boolean value,* `0` *representing* `true` *and other values representing* `false`*. The following sequence of commands illustrates a use of the* `if .. then .. [else ..] fi` *construction.*

```
touch zzz     # to guarantee file zzz exists
if test -f zzz
then
    echo "File zzz exists"
else
    echo "File zzz does not exist"
fi
check()
{
    if test -f $1
    then
        echo -e "\e[33mFile zzz exists\e[0m"
    else
        echo -e "\e[31mFile zzz does not exist\e[0m"
    fi
}
touch zzz     # to guarantee file zzz exists
```

```
check zzz
rm -f zzz     # to guarantee file zzz does not exist
check zzz
```

3. *Function* `test` *can be called using brackets.*

```
check()
{
    if [ -f $1 ]
    then
        echo -e "\e[33mFile zzz exists\e[0m"
    else
        echo -e "\e[31mFile zzz does not exist\e[0m"
    fi
}
touch zzz     # to guarantee file zzz exists
check zzz
rm -f zzz     # to guarantee file zzz does not exist
check zzz
```

4. *Operator* `!` *is the logical not.*

```
rm -f zzz     # to guarantee file zzz does not exist
if ! test -f zzz
then
    echo "File zzz does not exist"
fi
```

5. *Operators* `&&` *and* `||` *are simplified conditional constructions.*

```
touch zzz     # to guarantee file zzz exists
test -f zzz && echo "File zzz exists"
rm -f zzz     # to guarantee file zzz does not exist
test -f zzz || echo "File zzz does not exist"
```

Exercise 8 *The multiple choice* `case` *construction.*

1. *The* `case` *construction allows branching based on patterns, as is illustrated by the following code.*

```
z()
{
    case $# in
        0) echo "No arguments were given";;
        1) echo "One argument was given";;
        2|3) echo "Two or three arguments were given";;
        *) echo "More than three arguments were given";;
    esac
}
z
z aa
z aa bb
z aa bb cc
z aa bb cc dd
z aa bb cc dd ee
```

The double `;;` *is used to end a branch. The* `|` *in a pattern defines an alternative. The* `*` *means any value. In the previous code, being the last, it represents the otherwise values.*

Exercise 9 *The repetitive* `for` *construction.*

1. *The* `for` *construction allows for iteration through a list of values. Next code appends a prefix to the name of all files in the current directory whose names start with an* `a`*.*

```
touch a1 a2 a77 abc b1 c12 ddd    # create some files
ls
prefix="_a_"
for f in a*
do
    echo "changing the name of \"$f\""
    mv $f $prefix$f
done
ls
```

The following code creates and uses a function that iterates through all files passed as arguments.

```
f1()
{
    for file in $*
    do
        echo "==== $file ====" > $file
    done
}

f1 abc xpto zzz
cat xpto
cat abc
cat zzz
```

Exercise 10 *The repetitive* `while` *and* `until` *constructions.*

1. *In the following code, functions* `f2` *e* `f3` *are equivalent to function* `f1` *of the previous exercise.*

```
f2()
{
    while [ $# -gt 0 ]
    do
        echo "==== $1 ====" > $1
        shift
    done
}

f3()
{
    until [ $# -eq 0 ]
    do
        echo "==== $1 ====" > $1
        shift
    done
```

```
}

rm -f abc xpto zzz   # to guarantee they do not exist
f2 abc xpto zzz
cat xpto
cat abc
cat zzz
f3 abc xpto zzz
cat xpto
cat abc
cat zzz
```

Exercise 11 *Script files.*

1. *You can create a file whose contents is a program in* `bash` *(or a program in any other scripting language). To execute it, you can pass its name as an argument to the* `bash` *command or change its permissions to include execution. Such program are usually called* shell scripts. *Use your favorite text editor to create the file* `myscript`*, with the following contents.*

```
#!/bin/bash
# The previous line (comment) tells the operating system that
#   this script is to be executed by bash
#
# This script selects and sorts the lines of a given file,
#   except the first 5 and the last 5.
#
if [ $# -ne 1 ]
then
    echo "A single argument is mandatory" 1>&2
    exit 1
fi

if ! [ -f $1 ]
then
    echo "Given argument ($1) is not a regular file" 1>&2
    exit 1
fi

head -n -5 $1 | tail -n +6 | sort
```

2. *In the following code, the previous script is called through the* `bash`*.*

```
bash myscript
bash myscript xpto abc for testing
rm -f abc       # to guarantee it does not exist
bash myscript abc
# create a file for testing
seq -w 100 -3 2 > xpto
bash myscript xpto
```

3. *In the following code, permissions are changed to include execution and thus the script can be called directly. Actually, the code is executed within a new* `bash`*, implicitly called.*

```
chmod +x myscript
./myscript xpto
```

Exercise 12 `Bash` *supports both indexed and associative arrays.*

1. *The indices of an indexed array do not need to be contiguous and can not be negative. The following code shows the use of an indexed array.*

```
a[1]=aaa
echo ${a[1]}
declare -a a[2]=bbb  # can also be used
a[4]=ddd
a[2+3]=eee           # integer arithmetic expression are allowed
echo ${a[*]}         # the list of elements in the array
echo ${#a[*]}        # the number of elements in the array
echo ${!a[*]}        # the list of indices used in the array

# iterate through the list of elements
for v in ${a[*]}
do
    echo $v
done

# iterate through the list of indices
for i in ${!a[*]}
do
    echo "a[$i] = ${a[$i]}"
done
```

2. *Associative arrays need to be declared explicitly. The following code illustrates its declaration and use.*

```
declare -A arr
arr["homem"]=man
arr["papel"]=paper
arr["olá"]=hello
arr["lição"]=lesson
echo ${arr[*]}       # the list of elements in the array
echo ${#arr[*]}      # the number of elements in the array
echo ${!arr[*]}      # the list of indices used in the array
for i in ${!arr[*]}
do
    echo "The translation of \"$i\" is \"${arr[$i]}\""
done
```