Bash Scripting

Antonio Mucherino

www.antoniomucherino.it

University of Rennes 1, Rennes, France

last update: May 2nd 2016

What's *bash*

Bash is a shell written by Brian Fox for the GNU Project as a free software.

Bash is widely distributed as a default shell for Linux and Mac OS X.

Bash is a command processor, and it typically runs in a text window.

Bash allows the user to type commands which cause actions.

Bash can also read commands from files, which are called scripts.

Bash has programming language features (interpreted).

Why *bash*

It's the standard GNU shell, intuitive and flexible.

Other shells:

- **sh** subset of Bash, basic shell originally developed for Unix.
- **csh** a shell whose syntax resembles to the C programming language.
- **tcsh** superset of **csh**, enhancing user-friendliness and efficiency.
- **ksh** superset of **sh**, for experts.

The main aim of this course is to learn how to efficiently use and develop scripts in Bash.

Before continuing . . .

It is very important that you're familiar with the concept of

algorithm

Why *bash*

It's the standard GNU shell, intuitive and flexible.

Other shells:

- **sh** subset of Bash, basic shell originally developed for Unix.
- **csh** a shell whose syntax resembles to the C programming language.
- **tcsh** superset of **csh**, enhancing user-friendliness and efficiency.
- **ksh** superset of **sh**, for experts.

The main aim of this course is to learn how to efficiently use and develop scripts in Bash.

Before continuing . . .

It is very important that you're familiar with the concept of

algorithm

Let's get started: Hello World!

The first bash script

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

- all lines starting with `#` are considered as comments, and won't be interpreted
- the line "`#!/bin/bash`" must be present in every bash script, at the beginning of the file
- `echo` is a bash shell command that prints on the screen its arguments

Hello World: execution

In order to execute the script, we need to

- copy its content into a text file: `hw`
- make sure we have the right to execute the file:

```
-rwxr-x--x 1 mucherin genscale 75 Sep 11 14:41 hw*
```

- invoke the script:

```
> hw
Hello World!
>
```

Variables

In Bash, we can use variables for holding certain values

- integer numbers (not real ones in Bash)
- boolean values (as integers, e.g. 0 and 1)
- strings (ordered sets of characters)

in the computer memory.

Variables

Variable: a symbol representing a quantity capable of assuming any of a set of values.

Example: `number=1`

Important: Never leave a blank character between the variable name and the assignment symbol "=".

The access to a variable content can be done by using the symbol "`$`"

```
number=1     echo $number
```

Arrays of variables

An ordered list of variables can be represented by an array in Bash:

Integer variables	Strings
`i=1 ; a[$i]=$i`	`str[1]='first string'`
`i=2 ; a[$i]=$i`	`str[2]='second string'`
`i=4 ; a[$i]=$i`	`str[4]='last string'`

Indices do not have to be consecutive.

Array information:

`${arr[*]}` `${arr[@]}`	refers to all items in the array
`${!arr[*]}` `${!arr[@]}`	refers to all indices in the array
`${#arr[*]}` `${#arr[@]}`	is the number of items in the array
`${#arr[i]}`	is the length of item `i`

Bash does not support multidimensional arrays.

Input arguments

Previously defined variables can be given to bash scripts as input arguments:

- **$#** – this symbol refers to the number of input arguments (it does not count **$0**)
- **$0** – this symbol refers to the name of the script (this is always the first argument)
- **$1** – this symbol refers to the second input argument, if any
- **$2** – ... ...
- **${**10**}** – this symbol refers to the eleventh input argument, if any
- ... ...
- **$*** or **$@** – these two symbols refer to all input arguments, in order, starting from the argument **$1**

Input arguments

A set of arguments can be accessed in different ways.

Possibility 1 We use the symbol **$@** (or the symbol **$***) to refer to a string (only one!) containing all arguments (starting from $1)

Possibility 2 We use the symbol **$@** (but not the symbol **$***), and we copy the whole set of arguments inside an array:

```
array=("$@")
```

Possibility 3 We use the command `shift`:

$#** ⟵ **$#-1

lost ⟵ **$1** ⟵ **$2** ⟵ **...** ⟵ **$9** ⟵ **${10}** ⟵ **...**

A personalized message

The script:

```
#!/bin/bash
# personalized Hello World script
echo "Hello $1 $2 !!!"
```

The execution (we still suppose this script is the text file “hw”)

```
>
> hw Nicolas Sarkozy
Hello Nicolas Sarkozy !!!
>
```

Question: what if we expect more arguments (more pairs *first name* / *surname*) ??

A personalized message

```
#!/bin/bash
# personalized Hello World script
echo "Hello $* !!!"

> hw Nicolas Sarkozy
Hello Nicolas Sarkozy !!!
>
> hw Francois Hollande
Hello Francois Hollande !!!
>
> hw Nicolas Sarkozy Francois Hollande
Hello Nicolas Sarkozy Francois Hollande !!!
>
```

Question: what if we want to make the message nicer, by separating the names and surnames with the word “and” ??

A personalized message

```
#!/bin/bash
# personalized Hello World script
echo -n "Hello "
while [ $# -gt 0 ]
do
   echo -n $1; shift
   echo -n " "
   echo -n $1; shift
   if [ $# -gt 0 ]
   then
      echo -n " and "
   fi
done
echo " !!!"

>
> hw Nicolas Sarkozy Francois Hollande
Hello Nicolas Sarkozy and Francois Hollande !!!
>
```

Notice the use of the option `-n` in `echo`.

Control structures

In the last example, we made use of control structures.

IF executes a command (or a block of commands) when a certain condition is satisfied

FOR repeats a command (or a block of commands) a predefined number of times

WHILE executes a command (or a block of commands) *while* a given condition is satisfied

REPEAT . . . UNTIL executes a command (or a block of commands) *until* a given condition is satisfied

The sum of *n* numbers

Text file `sum`

```
#!/bin/bash
sum=1
for (( i=2 ; i<=$1 ; i++ ))
do
   sum=$sum+$i
done
echo "The sum of the first $1 integer numbers is $sum"
```

Execution:

```
> sum 5
The sum of the first 5 integer numbers is 1+2+3+4+5
```

Note that, by default, the arithmetic operation '+' is **not** executed!!

We need to use the command **let**.

The sum of *n* numbers

Text file `sum` with the command `let`

```
#!/bin/bash
sum=1
for (( i=2 ; i<=$1 ; i++ ))
do
   let sum=$sum+$i
done
echo "The sum of the first $1 integer numbers is $sum"
```

Execution:

```
> sum 5
The sum of the first 5 integer numbers is 15
> sum 100
The sum of the first 100 integer numbers is 5050
```

Prime numbers

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

```
 1:  #!/bin/bash
 2:  let n=$1       # number to check
 3:  let m=1+$n/2   # divisor cannot be greater than n/2
 4:  let bool=1     # true
 5:  for (( i=2 ; i<=m; i++ ))
 6:  do
 7:     let d=$n%$i
 8:     if [ $d -eq 0 ]
 9:     then
10:        let bool=0  # false
11:     fi
12:  done
13:  if [ $bool -eq 1 ]
14:  then
15:     echo "$n is a prime number"
16:  else
17:     echo "$n is NOT a prime number"
18:  fi
```

Prime numbers

Explanation of main lines:

`2:` the number to be checked is copied in `n`

`3:` the script tries to divide `n` by all integers smaller than `n/2`

`7:` the operator `%` gives the rest of the division of `n` by `i`, with `i` having values from `2` to `m`

`8:` if one of the rests `d` is 0, then `n` admits a positive divisor greater than 2, and therefore it is not a prime

`8:` **important:** in **if** structures, always leave a blank character between the conditions and the brackets

`13:` the information “not prime” is saved in a boolean variable, that is reused at the end of the script for printing the appropriate message

Boolean variables and operators

There is no specific type for boolean variables in Bash.

True	False
1	0

Comparisons in bash:

integer numbers

-eq	-ne	-lt	-gt	-le	-ge
$=$	$\neq$	$<$	$>$	$\leq$	$\geq$

strings

-z	-n	$=$	!$=$
is empty	is not empty	$=$	$\neq$

Logical operations in bash:

-a	-o	!
and	or	negation

Prime numbers

Some executions:

```
> prime 21
21 is NOT a prime number
>
> prime 17
17 is a prime number
>
> prime 121
121 is NOT a prime number
>
> prime 27
27 is NOT a prime number
>
> prime 31
31 is a prime number
>
> prime 87
87 is NOT a prime number
>
> prime 11
11 is a prime number
>
```

A detailed execution

Let's execute the script for `n = 21`.

Step 1. we set `m = 11`

Step 2. we divide `m` by `i`, for each `i` from 2 to `m`:

i	2	3	4	5	6	7	8	9	10	11
d	1	0	1	1	3	0	5	3	1	10

Step 3. `n` is not prime.

At the second iteration of the for loop, we can already state that the number is not prime!

Prime numbers

An improved version with **WHILE**.

```
 1:  #!/bin/bash
 2:  let n=$1       # number to check
 3:  let m=1+$n/2   # divisor cannot be greater than n/2
 4:  let bool=1     # true
 5:  let i=2
 6:  while [ $i -le $m -a $bool -eq 1 ]
 7:  do
 8:     let d=$n%$i
 9:     if [ $d -eq 0 ]
10:     then
11:        let bool=0  # false
12:     fi
13:     let i=$i+1
14:  done
15:  if [ $bool -eq 1 ]
16:  then
17:     echo "$n is a prime number"
18:  else
19:     echo "$n is NOT a prime number"
20:  fi
```

Prime numbers

. . . and **REPEAT . . . UNTIL**.

```
 1:  #!/bin/bash
 2:  let n=$1       # number to check
 3:  let m=1+$n/2   # divisor cannot be greater than n/2
 4:  let bool=1     # true
 5:  let i=2
 6:  until [ $i -gt $m -o $bool -eq 0 ]
 7:  do
 8:     let d=$n%$i
 9:     if [ $d -eq 0 ]
10:     then
11:        let bool=0  # false
12:     fi
13:     let i=$i+1
14:  done
15:  if [ $bool -eq 1 ]
16:  then
17:     echo "$n is a prime number"
18:  else
19:     echo "$n is NOT a prime number"
20:  fi
```

Manipulating the file system

All commands available in Bash can be used in the scripts

- ls
- pwd
- cd
- cp
- mv
- rm
- mkdir
- cat
- who
- find
- grep
- ps
- alias
- chown
- make
- tar
- ar
- ...

Combining commands in scripts

Commands `pwd` and `ls`, in a single shot:

```
> cat pwdls
#!/bin/bash
echo "The content of the directory:"
pwd
echo "is the following:"
ls
>
> ls
prime*  prime2*  pwdls*
> pwd
/userfiles/Bash/scripts
>
> pwdls
The content of the directory:
userfiles/Bash/scripts
is the following:
prime*  prime2*  pwdls*
>
```

Conditions concerning files

Conditions to be verified in **if** control structures can also concern files in the file system.

```
if [ option $file ]
then
    ...
fi
```

option	verifies whether
-e	the file exists
-f	it's a normal file
-d	it's a directory
-r	it can be read
-w	it can be modified
-x	it can be executed
-s	it's not empty

`file` is a string containing the name of the file to be checked.

For loop for files

The **for** control structure can be used for enumerating all files (or part of them) belonging to a given directory.

```
> ls
myls*  prime*  prime2*  pwdls*
>
> cat myls
#!/bin/bash
for i in *
do
   echo $i
done
>
> myls
myls
prime
prime2
pwdls
>
```

sl : an inverted ls

In our scripts, we may want to consider the files in an inverse alphabetic order.

Script:

```
#!/bin/bash
let k=0
# copying
for i in *
do
   let k=$k+1
   file[k]=$i
done
let n=$k
# printing in the inverse sense
for (( k=$n ; k>0 ; k-- ))
do
   echo ${file[k]}
done
```

Execution:

```
> myls
myls
prime
prime2
pwdls
sl
>
> sl
sl
pwdls
prime2
prime
myls
```

The magic dollar $

The symbol **$** can be used for accessing the value of a variable, as well as the value of the input parameters of a script.

The symbol **$** can also be used for retrieving the output from commands that are executed inside scripts (standard output)

```
> cat dollar1
var=$(whoami)
echo "My username is: $var"
> dollar1
My username is: mucherin
>
> cat dollar2
var=$(pwd)
echo "The current directory is: $var"
> dollar2
The current directory is: /userfiles/Bash/scripts
>
```

The size of a file

du is one of the most reliable commands for retrieving the size (in terms of bytes with the option **-b**) of a file. However, it also provides additional information we may not be interested in.

```
> du -b BashScripting.pdf
311119  BashScripting.pdf
```

The output is given in tabular format (TAB separates the size and the name). The command **cut** can be used for filtering the fields of a table:

```
> du -b BashScripting.pdf | cut -f 1
311119
```

Finally, in a script, we may write:

```
var=$(du -b BashScripting.pdf | cut -f 1)
```

The **m**aximum **ls** (**mls**)

Let's write a script that only prints the largest file of a given set of files:

```
> ls -l
-rw-r----- 1 mucherin genscale 416125 Sep 26 15:53 BashScripting.pdf
-rw-r----- 1 mucherin genscale  25508 Sep 26 15:56 BashScripting.tex
-rw-r----- 1 mucherin genscale    175 Sep 11 14:21 makefile
>
> mls *
416125   BashScripting.pdf   1
>
```

The list of files is given as an input argument (the symbol * is here used for considering all files in the current directory).

The **m**aximum **ls** (**mls**)

```
#!/bin/bash
maxsize=0
posmax=0
filename="(no files found)"
# identifying the file with maximum size
k=0
while [ $# -gt 0 ]
do
   if [ -f $1 ]
   then
      let k=$k+1
      size=$(du -b $1 | cut -f 1)
      if [ $maxsize -lt $size ]
      then
         filename=$1
         maxsize=$size
         posmax=$k
      fi
   fi
   shift
done
echo -e "$maxsize \t $filename \t $posmax"
```

Notice that `echo` is invoked with the option `-e`, which allows the use of special characters, such as TAB (`\t`).

du and mls

These two commands have a similar output format:

```
> du -b BashScripting.pdf
371814 BashScripting.pdf
>
> ls
BashScripting.pdf BashScripting.tex makefile
>
> mls *
371814 BashScripting.pdf 1
```

They both output the results in tabular format.

Single brackets [·]

Single brackets [·] are equivalent to the execution of the command `test`:

```
if [ $a -eq $b ]        if test $a -eq $b
```

The second syntax works because `test` gives as an input an integer number:

$= 0$ if the condition is satisfied,
$\neq 0$ otherwise.

Since other commands and programs work at the same way (they return 0 if they were executed with success), then the **if** control structure can also be used as follows:

```
if cp $filename backup
then
   echo "copy of $filename saved"
else
   echo "impossible to copy file $filename"
fi
```

Double parenthesis ((·))

Double parentheses allow us to perform arithmetic operations (like `let`) while omitting the dollar `$` and enabling to include blank characters around operators (improving thus readability).

```
let a=$b+1          a = $((b + 1))
```

Double parentheses are used in the for control structure: this is the reason why it is not necessary to use `$` when referring to the variables:

```
for (( i=1 ; i<n ; i++ ))
do
   ...
done
```

Double parentheses allow for using a C-like syntax:

```
((a = 10))     ((i++))     if ((a == 10))
```

Double brackets [[·]]

In more recent implementations of Bash, double square brackets are an extension of single ones, where C-like operators (such as `&&` and `||`) are allowed.

```
if [ $a -eq 0 -a $b -ne 1 ]
then
   ...
   ...
fi
```

```
if [[ $a == 0 && $b != 1 ]]
then
   ...
   ...
fi
```

When using double brackets, strings containing blank characters are *not* separated in different words!

Single braces { · }

Braces allow for generating an ordered list of integer numbers or characters in an easy and intuitive way.

```
for i in {1..10}
do
   echo $i
done
```

```
for (( i=1; i<=10; i++ ))
do
   echo $i
done
```

Other examples:

```
var1=1
var2=5
echo {e..m}                          #  e f g h i j k l m
echo {3..-2}                         #  3 2 1 0 -1 -2
echo {1..2}{x..y}" +" + "..."        #  1x + 1y + 2x + 2y + ...
echo {$var1..$var2}                  #  1 2 3 4 5
```

Remember that we already used braces when working with arrays in Bash.

Avoiding expansions

The first interpretation task of the shell is the so-called *expansion* of the special characters, including brackets. Other examples are:

*	all files in the current directory
.	the current directory
!	last command in the shell

Important: expansions also apply to characters contained in strings! To avoid this:

use backslash	*	for one character only
use quotes	**“yes!”**	for more than one character

String manipulation operators

Let's consider this short Italian sentence: `str="ciao bella"`

This is a small list of manipulation operations that can be applied to strings:

`n=${#str}`	`10`
`a=${str:0:4}`	`ciao`
`b=${str:(-5)}`	`bella`
`c=${str#ciao}`	`bella`
`c=${str#c*o}`	`bella`
`d=${str%bella}`	`ciao`
`d=${str%b*a}`	`ciao`
`e=${str/bella/brutta}`	`ciao brutta`

`#` → removes the prefix that follows this symbol (if present)
`%` → removes the suffix that follows this symbol (if present)
`*` → refers to any substring
the command `sed` performs similar operations on text files

Patterns and comparisons

In some applications, we may need to compare our strings to a certain number of possible strings.

For example, the string `[Hh]ello World!` matches with both

`Hello World!` `hello World!`

Pattern generators:

`[abc]`	matches with either `a`, `b` or `c`
`[^abc]`	negation of what above
`[a-z]`	matches with all characters from `a` to `z`
`[1-9]`	matches with all numbers in the given range

Syntax for performing the comparison:

```
if [[ $substring =~ [abc]ed ]] then do ... done
```

The command **bc**

bc is an arbitrary precision calculator language.

```
> bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Softwa
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
a = 1   # I wrote this
b = 2   # I wrote this
a+b     # I wrote this
3
a-b     # I wrote this
-1
a/b     # I wrote this
0
scale=10    # I'm changing the precision
a/b         # trying again ...
.5000000000
quit
```

Average file size

If we can deal with real numbers, in our scripts we can compute, for example, the average of a set of integer numbers.

```
> ls -l
total 24
-rwxr-x--- 1 mucherin genscale 416 Sep 29 23:46 minmaxavg*
-rwxr-x--- 1 mucherin genscale 380 Sep 23 17:22 mls*
-rwxr-x--- 1 mucherin genscale 483 Sep 23 17:22 sls*
-rwxr-x--- 1 mucherin genscale 635 Sep 29 23:45 sortls*
-rwxr-x--- 1 mucherin genscale 681 Sep 29 23:45 sortls2*
-rwxr-x--- 1 mucherin genscale 994 Sep 29 23:45 sortls3*
> minmaxavg *
380       606.1666666666          994
>
```

Average file size

```
#!/bin/bash
size=$(du -b $1 | cut -f 1) ; n=1
min=$size ; max=$size ; sum=$size
shift
while [ $# -gt 0 ]
do
   if [ -f $1 ]
   then
      let n=$n+1
      size=$(du -b $1 | cut -f 1)
      if [ $min -gt $size ]
      then
         min=$size
      fi
      if [ $max -lt $size ]
      then
         max=$size
      fi
      sum="$sum + $size"
   fi
   shift
done
avg=$(echo "scale=10; ($sum)/$n" | bc)
echo -e "$min \t $avg \t $max"
```

Functions in Bash

Every time it is necessary to repeat (in different parts of a script, and/or in different scripts) the same set of commands, we can create a function containing such a set of commands:

```
function simplefun ()
{
   echo "My first argument is: $1; "
   echo "My second argument is: $2"
}
```

Input/output in functions:

- input arguments are given to the function as in scripts: the same set of built-in variables can be used,
- the function returns its output to the invoking script by using the command `echo`.

Functions in Bash

Consider the following script (simplefun appears before its call):

```
#!/bin/bash
function simplefun ()
{
   echo "My first argument is: $1; "
   echo "My second argument is: $2"
}
var=$(simplefun $*)
echo $var
```

In order to invoke a function that is contained in another script:

```
#!/bin/bash
source script_containing_simplefun.sh
var=$(simplefun $*)
echo $var
```

What do you expect to be the output on the screen?

```
> testfun 1 2
My first argument is: 1; My second argument is: 2
```

Functions in Bash

Consider the following script (simplefun appears before its call):

```
#!/bin/bash
function simplefun ()
{
   echo "My first argument is: $1; "
   echo "My second argument is: $2"
}
var=$(simplefun $*)
echo $var
```

In order to invoke a function that is contained in another script:

```
#!/bin/bash
source script_containing_simplefun.sh
var=$(simplefun $*)
echo $var
```

What do you expect to be the output on the screen?

```
> testfun 1 2
My first argument is: 1; My second argument is: 2
```

Local and global variables

Differently from many programming languages, a variable that is used inside a function is visible everywhere else (other functions, the main script).

In order to force a certain variable to be *local*, we can employ the following syntax:

```
glo_var=100    # this is a global variable
local loc_var=200    # this is a local variable
```

Therefore, *it's necessary paying attention to variables having the same name that might be used in various functions and/or in the main script*.

Another syntax for **for**

The control structure **for** can work on arrays of strings, as well as on strings containing more items separated by blank characters.

```
let k=1
for i in *
do
   allfiles="$allfiles $i"
   file[k]=$i
   let k=$k+1
done
```

```
for i in ${file[@]}
do
   echo $i
done
```

```
for i in $allfiles
do
   echo $i
done
```

Reading data

The command `read` is able to load inside a variable the content of a string of characters:

- it can be written at the prompt by using the keyboard
- it can be redirected from a text file by using the shell
- it can be redirected from a text file inside a script

```
> cat myscript
...
while read line
do
   ... ... ...
done
...
>
> myscript < textfile.txt
...
> cat textfile.txt | myscript
...
>
```

```
> cat myscript
...
while read line
do
   ... ... ...
done < $filename
...
>
> myscript
...
```

Selecting the rows of a table

Let us write a script that selects the rows of a table having a certain property.

```
> cat phonenumbers.txt
Alain Delon M 0033 123456789
Nicole Kidman F 001 123456789
Francesca Neri F 0039 123456789
Tom Hanks M 001 123456789
Terence Hill M 0039 123456789
Eva Herzigova F 00420 123456789
Hugh Laurie M 0044 123456789
>
> select M < phonenumbers.txt
Nicole Kidman F 001 123456789
Francesca Neri F 0039 123456789
Eva Herzigova F 00420 123456789
>
```

In this example, all rows referring to phone numbers belonging to woman are selected ;-)

Selecting the rows of a table

This is the script:

```
#!/bin/bash
sex=$1
while read -a arr
do
   if [ ${arr[2]} != $sex ]
   then
      echo ${arr[*]}
   fi
done
```

Remarks:

- `sex` is an input argument
- the option `-a` for `read` indicates that the input text is supposed to be separated in words (by default, the separation field is " ")
- `arr` is therefore an array

The king of text processing

AWK is a simple and fast command for text processing.

- it is meant to work on column-oriented text data, such as matrices and tables
- it also has some programming language features

This is the simulation of `cat` with `awk`:

```
awk '{print $0}' textfile
```

- `$0` refers to the generic line of the file
- all instructions between `{` and `}` are executed for each line of the file

The king of text processing

Built-in variables in `awk`:

NR	counter of records (lines in the text file)
NF	number of fields (words per line)
FS	field separator (the character between two words, default is " ")
$0	the current line (entire)
$1	the first word in the current line
$2	the second word in the current line
...	...
$($i$)	the i^{th} word in the current line
...	...
$(NF-1)	the last but one word in the current line
$NF	the last word in the current line

The king of text processing

Control structures in `awk` *with some examples*:

```
> cat matrix.txt
1 -2  3  3
3  4 -5  3
5  4  2 -7
>
> awk '{if ($1 > 2) print $0}' matrix.txt
3  4 -5  3
5  4  2 -7
>
> awk '{ for (i = 1; i<=NF; i++) {
    if ($i < 0) { printf "%d ",-$i } else { printf "%d ",$i } }
        printf "\n" }' matrix.txt
1 2 3 3
3 4 5 3
5 4 2 7
>
```

Notice that `printf` allows us to print in the format we prefer (it is similar to the C function and to the Bash command `printf`)

The king of text processing

Let's apply this awk script to the list of our phone contacts:

```
awk '{if ($3 != "M") print $0}' phonenumbers.txt
```

or in short

```
awk '$3 != "M"' phonenumbers.txt
```

This is the result:

```
Nicole Kidman F 001 123456789
Francesca Neri F 0039 123456789
Eva Herzigova F 00420 123456789
```

Exactly the same we obtained with a Bash script!!

Selecting rows or columns

This short awk script selects the rows of a matrix having pair index:

```
awk 'NR%2 == 0' matrix.txt
```

This short awk script selects the columns of a matrix having pair index:

```
awk '{
        for (i=1; i<=NF; i++)
        {
           if ($i%2 == 0) printf "%s ",$i
        }
        printf "\n"
     }' matrix.txt
```

BEGIN and END

By default, **awk** processes the input text files line by line.

However, we might need to execute some actions *before* this process begins, or *after* it ends.

BEGIN all actions are executed *before* processing the lines of the text file

END all actions are executed *after* processing the lines of the text file

```
awk 'BEGIN { begin actions }
           { actions line by line }
      END { last actions } '
```

BEGIN and END

A simple example with **BEGIN** and **END**:

```
> awk '{ print "beginning" ; print $0 ; print "ending" }'
      matrix.txt
beginning
1 -2  3  3
ending
beginning
3  4 -5  3
ending
beginning
5  4  2 -7
ending
>
> awk 'BEGIN { print "beginning" }
             { print $0 }
         END { print "ending" }' matrix.txt
beginning
1 -2  3  3
3  4 -5  3
5  4  2 -7
ending
>
```

Word counter

This awk script counts the words contained into text files:

```
awk '{
        for (i=1; i<=NF; i++) freq[$i]++
     }
 END {
        for (word in freq)
             printf "%s\t%d\n",word,freq[word]
     }' text.txt
```

Remarks:

- as in bash, there is a special syntax for the control structure **for** that allows us to iterate on the elements of an array
- in awk, indices of arrays can be strings !!!

Working on two text files

NR counts all lines, for each file; **FNR** counts lines of current file.

```
> cat matrix1.txt matrix2.txt
1 2
3 4
a b
c d
> awk '{
        if (NR==FNR)
        {
           print "first file :",$0
        }
        else
        {
           print "second file:",$0
        }
       }' matrix1.txt matrix2.txt
first file : 1 2
first file : 3 4
second file: a b
second file: c d
```

Working on two text files

Given two matrices a_{ij} and b_{ij}, having the same size, we want to define the matrix c_{ij} such that:

$$\forall i, j \quad c_{ij} = \max\{a_{ij}, b_{ij}\}.$$

```
> cat matrix1.txt
1 -2  3  3
3  4 -5  3
5  4  2 -7
> cat matrix2.txt
8  4 -5  3
1 -2  4  3
5 -4  1 -6
>
> awk -f maxelement.awk matrix1.txt matrix2.txt
8 4 3 3
3 4 4 3
5 4 2 -6
```

Working on two text files

The awk script:

```
{
    if (NR==FNR)
    {
        for (j=1; j<=NF; j++) var[FNR,j]=$j
    }
    else
    {
        for (j=1; j<=NF; j++) if (var[FNR,j] < $j) var[FNR,j]=$j
    }
}

END {
    for (i=1; i<=FNR; i++)
    {
        for (j=1; j<=NF; j++) printf "%d ",var[i,j]
        printf "\n"
    }
}
```

The field separator

By default, the field separator **FS** in awk is a blank character.

However, we can change it, or use more than one separator!

```
> date
Wed Nov 13 14:40:37 CET 2013
>
> date | awk '{print $1, $4}'
Wed 14:40:45
>
> date | awk -F ":" '{print $1, $4}'
Wed Nov 13 14
>
> date | awk -F "3 " '{print $1, $4}'
Wed Nov 1
>
> date | awk -F "[: ]" '{print $1, $4}'
Wed 14
>
```

awk one-liners

We print only the lines having at least one word:

```
awk 'NF > 0' file.txt
```

We print only lines starting with "ATOM"

```
awk '$1 == "ATOM"' file.txt
```

We add the line number at the beginning of each line

```
awk '{print NR, $0}' file.txt
```

We compute the total size of the current directory

```
ls -l | awk '{ x=x+$5 } END { print x }'
```

We remove the 2^{nd} column from a matrix

```
awk '{$2="", print $0}' file.txt
```

The End