Introduction to Bash programming/scripting

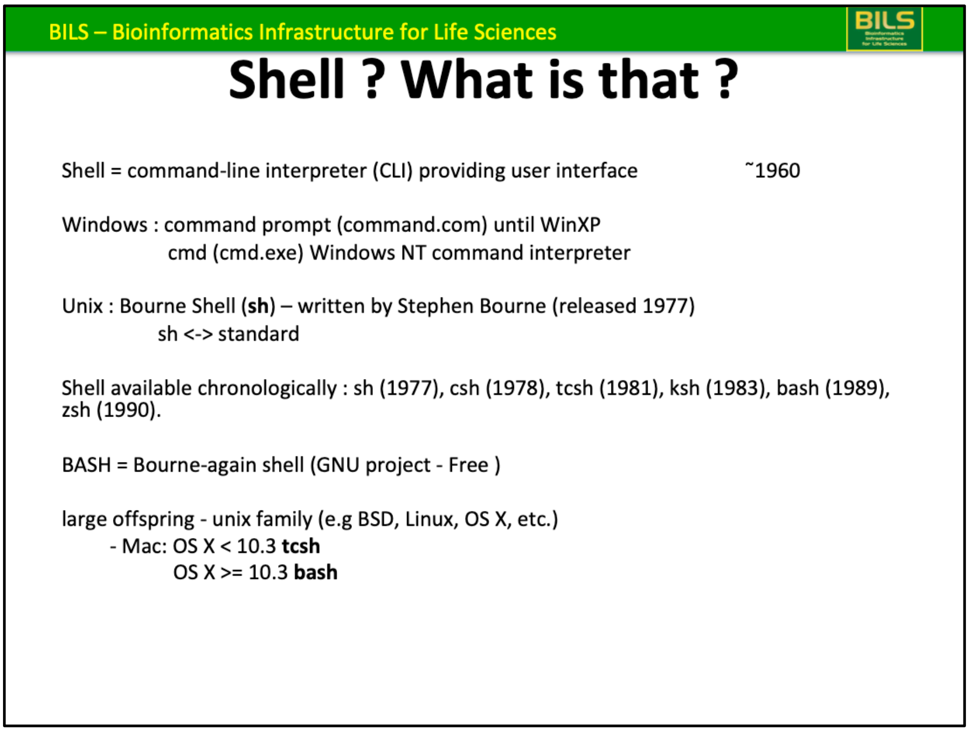
Jacques Dainat

2015

Summary

1) **Shell ? What is that ?**
2) **Bash? What is that ?**
3) **Bash tricks**
4) **I/O and Redirection**
5) **Scripting in bash**
6) **Examples**

! Commands are in green (most of time)



CLI occurred at the same time as the keyboard.
Windows and Unix are operating systems
Command prompt is often called MS-DOS or / DOS that is in reality the Operating system name.
The Bourne shell was one of the major shells used in early versions of the Unix operating system and became a de facto standard.

BILS – Bioinformatics Infrastructure for Life Sciences

Shell ? What is that ?

List all the available shells :

cat /etc/shells

/bin/bash
/bin/sh
/bin/tcsh
/bin/csh

Check your default shell:

echo $SHELL

To switch from one shell to another, just enter the name of the new shell in the active terminal.

To know the current shell working, type: **echo $0**

Which Bash version ? **/bin/bash -version**

- OS X Yosemite (Bash 3.2)
- Bash 4 exists since 2009

associative arrays available since bash 4

BILS – Bioinformatics Infrastructure for Life Sciences

Bash? What is that ?

Why BASH ?

- By default on most used Unix machines (Ubuntu, OS X)
- It can run almost all Bourne scripts and POSIX compliant scripts
- Syntax simplified :

  Single **[]** are posix shell compliant condition tests.

  Double **[[]]** are an extension to the standard **[]** and are supported by bash and other shells (e.g. zsh, ksh).

  They support extra operations (as well as the standard posix operations). For example: **||** , **&&** and regex matching with **=~** (Perl syntax).
- Associative arrays (since version 4)
- Free (GNU project)

POSIX: Portable Operating System Interface, standard specify for compatibility with variants operating systems

BILS – Bioinformatics Infrastructure for Life Sciences

Bash tricks

The commands:

- Despite they are often intuitive you have to learn them.

- You may look the /bin and /usr/bin directories that contain all the commands.
  ll /bin
  ll /usr/bin

- Internet is your friend (e.g.):
  - OS X command line : http://ss64.com/osx/
  - Linux command line : http://ss64.com/bash/

- Have a “lazy dog”

!! Use **man**, **help** or **info** to see documentation of each command
man *command*

/bin essentially contains command require by the system for emergency repairs, booting
/usr/bin contains the rest
man, **help** or **info** In that corresponding prioritization

Bash tricks

General:

Environment variables hold values related to the current environment.

env

- **PATH**: It specifies the directories in which executable programs are located

~/.bashrc or **~/.profile** (file read when open a new shell)

- typically used to change prompts, set environment variables, and define shell procedures.

e.g: * modified the PS1 variable to customize the prompt
http://www.cyberciti.biz/tips/howto-linux-unix-bash-shell-setup-prompt.html

* add alias: alias ll='ls -lGrt'
alias milou='ssh *user*@milou.uppmax.uu.se'

* Modify or add environment variables

source ~/.profile #take in account the modification in current shell

-lGrt: l for long format ; G for enable colorized output ;t to sort by time modified (most recently first); r for Reverse order - the oldest entries first (newest last = bottom)

Bash tricks

General:

- Use the tabulation key for auto-completion !
- **egrep, fgrep** and **rgrep** are often available but their direct invocation is deprecated (Is provided to allow historical applications that rely on them to run unmodified). Instead use **grep –E, grep –F** and **grep –r**.
- Need of calculation, type **bc**
- Pattern matching != Globbing
  Both use Wildcards but the first is for text matching while the second for file names.
  /!\ Same Wildcard doesn't have the same meaning.
  e.g: *
  Pattern matching follows the Perl syntax.

Wildcards are also called metacharacters.
* The preceding item matches 0 or more times.
* Zero or more characters

Bash tricks

Command substitution: ` ***command*** `
$(*command*)

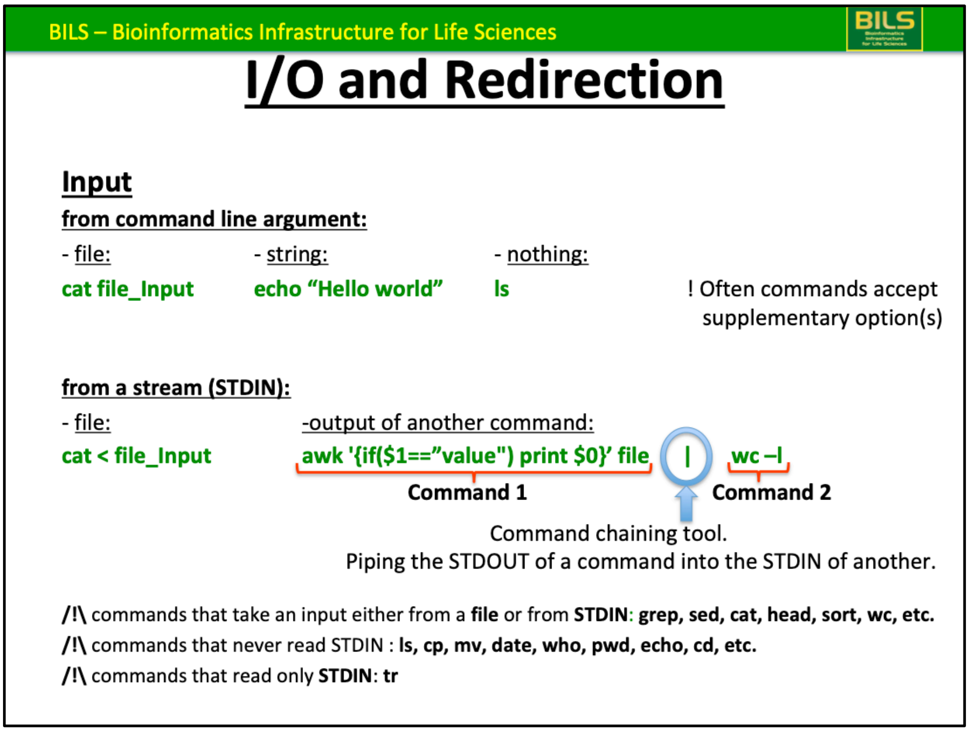
Bash replaces the command substitution with the standard output of the command.

The output of the command can be used in another command:

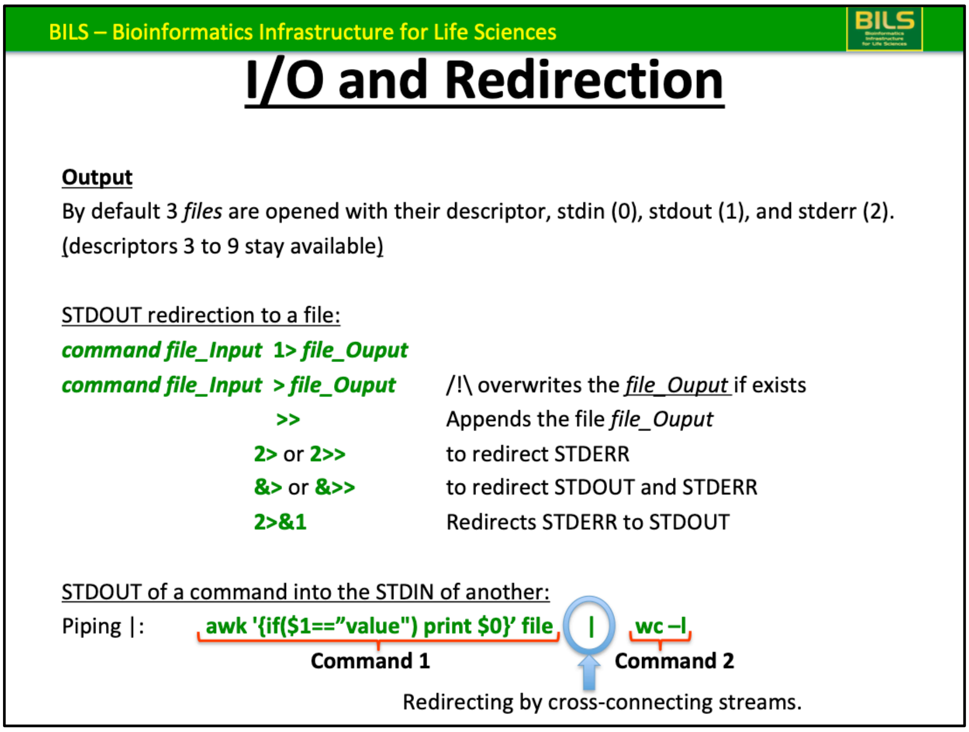
echo The current working directory is: `pwd`

or to set a variable:

var=$(pwd)



standard streams are preconnected input and output communication channels



Open a new descriptor: exec 3<file for reading (example with read: while read –u 3 line;do echo $line;done) - close it: exec 3<&-

: exec 3>file for writing - close it: exec 3>&-

Redirection tutorial: http://wiki.bash-hackers.org/howto/redirection_tutorial

BILS – Bioinformatics Infrastructure for Life Sciences

I/O and Redirection

Piping is powerful, but inappropriate if you need several command STDOUTs to feed the input of another command.

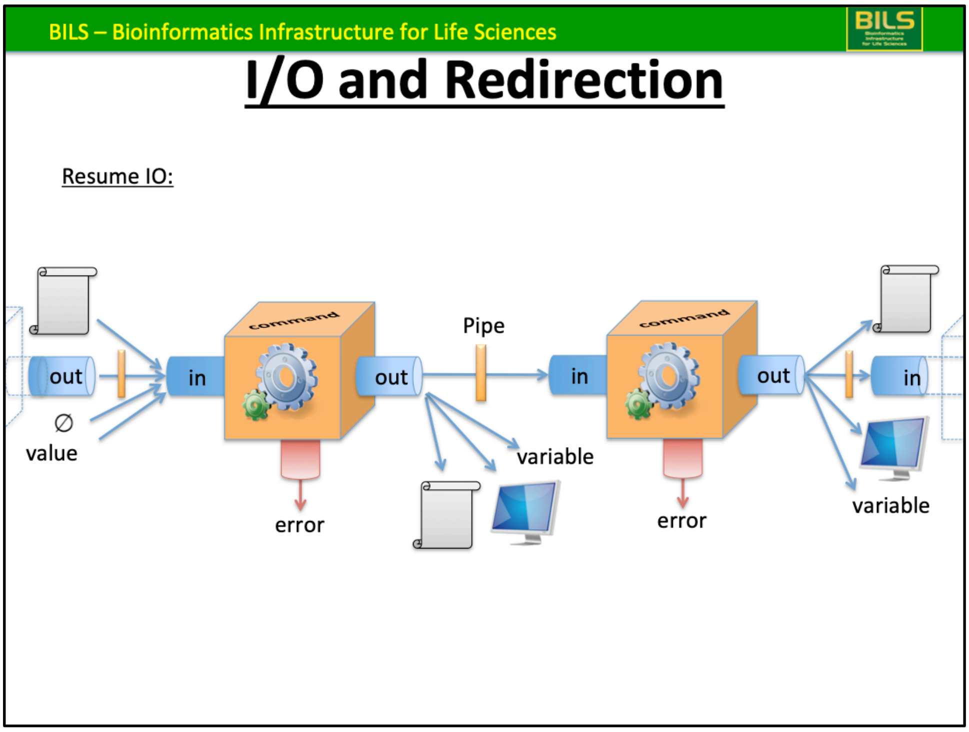
<u>Process substitution:</u> **<(*command*)**

Useful when a command needs a list of file as input.

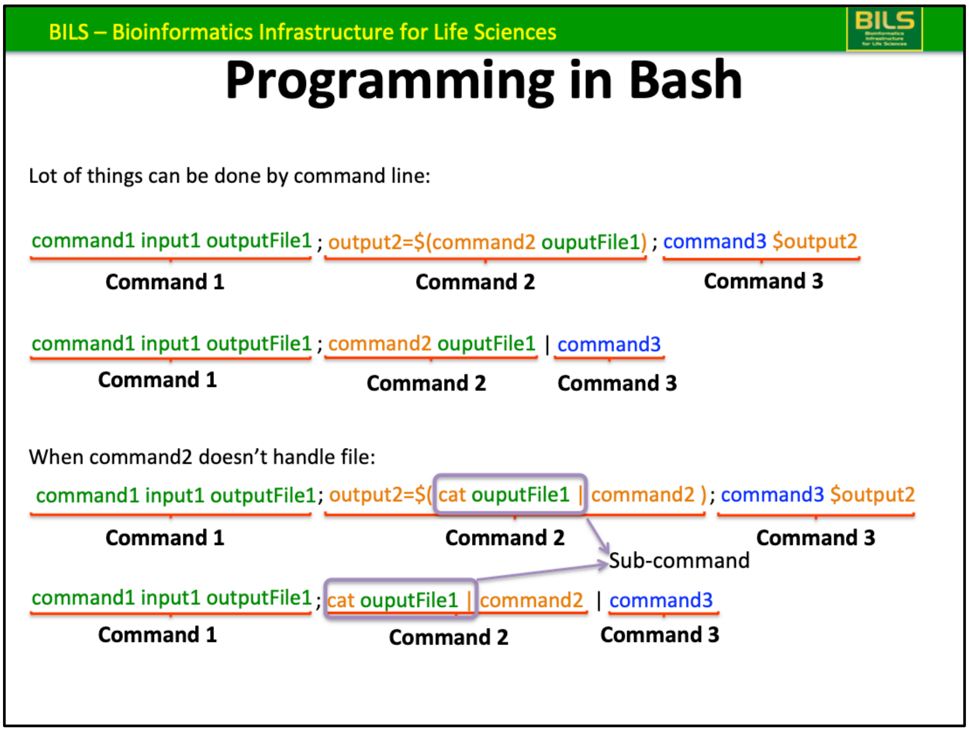
It generates a file.

diff <(ls $dir1) <(ls $dir2)

To check where the created temporary file is you can try: echo <(*command*)



Example of process substitution in output: *tar cfv >(bzip2 -c > dir1.tar.bz2) $dir1*



Same thing if command3 cannot Handle a STRING (as with **cut**) we should write: **; echo $output2 | cut –d''** **–f1**

Commands can contain control structures as Loop or if condition.

Programming in Bash

In command line or in a script you can use these syntaxes:

```
command1 ; command2
```
Several commands in a line

or

```
command1
command2
```
One command by line

```
command1 | command2
```
Several commands in a line with STDOUT redirection

or

```
command1 |
command2
```
One command by line with STDOUT redirection

BILS – Bioinformatics Infrastructure for Life Sciences

Programming in Bash

When used bash ?

/!\ Not adapted when:

- Resource-intensive tasks
- Complex application
- Need of specific library or data structure
- Cross-platform portability required

…

⇒ Great for small programs
⇒ More **scripting** rather than programming

Nice because:

- directly available via the terminal
- Easy to make script when you know command line. (automate routine)
- File handling
- System and Administrative
- Job Control

…

use *C* or *Java for cross platform portability*

Programming in Bash

Script or command line ?

Command line:

+ short task
+ specific task (Throw-away code)
- if long command => hard to debug ; hard to read
- variable may be mixed up with those setup previously
- no user interface (check of arguments, help, etc)

Script:

+ long tasks
+ re-usable
+ code structured => easy to read (comment, block of code, function)
+ more user friendly (Warning, check of parameters)
+ debug easier
- big collection of scripts may be annoying (lost time to find the good one)

Programming in Bash

How write a script ?

1) Open a file to write your script with **.sh** extension:

```
#!/bin/bash                                                  <- It's needed at the top of the file to specify the shell interpreter
echo "This is my first script" #display the sentence         <- One command (No semicolon needed)
# save the command in a variable, then print it              <- A comment
var=$(pwd); echo "my working directory is $var"              <- One command (No semicolon needed)
```

2) Save the file and give the execution right.

chmod 754 *myscript*.sh

3) Execute your script:

./*script*.sh

BILS – Bioinformatics Infrastructure for Life Sciences

Programming in Bash

2,5 primary data structures

Simple variables:

A variable in bash can contain a number, a character, a string of characters.

You have no need to declare a variable, just assigning a value to its reference will create it.

***variable*="a string with space"** or ***variable*=54** or ***variable*=$(command)**

Array:

array*=()** or ***array*=(Anna Par Ulla)** or **declare –a *array

/!\ The array is not initialized to empty in the last case if it already exists.

- Array index is an integer starting from 0.

Associative array (only with Bash>4):

declare –A *array* or ***array* =([*string1*]=*value1* [*string2*]=*value2*)**

/!\ problem if you try to do: ***variable*=54 toto** **<= bash try to execute «toto»**

-bash: toto: command not found

Programming in Bash

Array manipulation commands:

array[N]=value	Set the element *N* of the array *array* to *value*
array+=(value1 value2 value3)	Append the array with three values.
echo ${array[N]}	Display the element referenced by the index *N* from *array*.
echo ${#array[N]}	Display the length of the value referenced by the index *N* in *array*
echo ${#array[@]}	Display (number of elements) of *array*.
echo ${!array[@]}	Display each *array* index key as a separate argument.
echo ${array[@]}	Display all the values stored in *array*.
unset -v array[N]	Destroy the *array* element at index *N*.
unset -v array	Destroy the complete *array*.

This slide is really boring… I know

Programming in Bash

Calculation in bash

((*var = operation*)) or **var=$((*operation*))**
Assign the result of an arithmetic evaluation to the variable *var*.

/ !\ Natively Bash can only handle integer arithmetic.

Floating-point arithmetic:
You must delegate such kind of calcul to specific command line tool as **bc**.

echo "*operation*" | bc –l
Display the result of a floating-point arithmetic.
***var*=$(echo "*operation* " | bc -l)**
Assign the floating-point arithmetic result to the variable *var*.

Programming in Bash

Bash Control Structures 1) Conditional statements (on arithmetic values):

/ !\ The **spaces** are important in that syntax

```
If (( condition1 ));then
	command1
elif ! (( condition2 ));then
	command2
elif (( condition3 )) && (( condition4 ));then
	command3
elif (( condition5 ))  || (( condition4 )) ;then
	command4
else
	command5
fi
```

Logical operators are in green.

Programming in Bash

Bash Control Structures 1) Conditional statements (on string values):

/ !\ The **spaces** are important in that syntax

```
if [[ condition1 ]];then
    command1
elif ! [[ condition2 ]];then
    command2
elif [[ condition3 ]] && [[ condition4 ]];then
    command3
elif [[ condition5 ]]  || [[ condition4 ]] ;then
    command4
else
    command5
fi
```

Logical operators are in green.

Programming in Bash

Bash Control Structures 1) Conditional statements (next):

```
variable=$(command)
case $variable in
  pattern1)
    commands1
    ;;
  pattern2|pattern3|pattern4)
    commands2
    ;;
  patternN)
    commands3
    ;;
  *)
    commands4
    ;;
esac
```

Number of case infinite. It is a good alternative to if when lot of case to check.

Programming in Bash

Bash Control Structures 2) The loops:

A) The *for* loop:

Loop over list of elements (files or values):

```
for i in file1 file2 file3; do
    echo "this is one file $i"
done
```

```
for i in *.fasta; do
    command
done
```

Loop over file's lines:

```
for i in $(cat file.txt); do
    echo "this is one line: $i"
done
```

Loop over array:

```
for i in ${!array[@]} ; do
    echo "key :" $i
    echo "value:" ${array[$i]}
done
```

Programming in Bash

Bash Control Structures 2) The loops:

B) The *while* loop:

Loop over file's lines:

```
while read line ;do
    echo "this is one line: $line"
done < file
```

Loop over array:

```
i=0
while (( i < ${#array[@]} ));do
    echo "key :" $i
    echo "value:" ${array[$i]}
    ((i++))
done
```

=> It exists the useless **until** loop. Become a while loop by simply negating the condition.

Programming in Bash

Processes control :

ps ax	process status with a = show processes for all users x = show processes not attached to a terminal
jobs	List the active jobs
Fg	Switch a job running in the background into the foreground.
bg	Restart a suspended job, and run it in the background
Kill	Terminate a process
times	System times for processes run from the shell
Wait	Wait for the specified process and report its termination status….

Programming in Bash

What about library ?

Bash is quite limited but you can define a list of methods in a file. To include all the methods of this file in a script you have to write at the top of your script one of these lines (after the #!/bin/bash):

. /path/to/the/file

$include /path/to/the/file

source /path/to/the/file

More ?

AWK: http://www.grymoire.com/Unix/Awk.html

SED: http://www.grymoire.com/Unix/Sed.html

BASH: http://www.gnu.org/software/bash/manual/bashref.html
http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html
http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
http://www.tldp.org/LDP/abs/html/ (Advanced Bash-Scripting Guide)

For other Unix Shell commands or to compare them:
http://hyperpolyglot.org/unix-shells

Mac OS X version 10.9 Bash manual page:
https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/bash.1.html

A book ? => **bash Cookbook (O'Reilly)**

Examples

1) Reverse complementing

2) Get last two columns of a file

3) Remove isoform from proteome

4) Awk and join commands

5) Example of job control

6) downsampling a fastq

Correction:
Reverse complementing

echo *sequence* | rev | tr “ACGT” “TGCA”

Or

cat *file* | rev | tr “ACGT” “TGCA”

Correction:
Get last two columns of a file

awk ‘{print $NF”\t”$(NF-1)}’ *file*

Or

rev file | cut –f1,2 | rev

Correction:
Remove isoform from proteome

Remove isoforms:

```
#!/bin/bash

#This script kept only one isoform per gene for proteomes coming from Ensembl

# Arguments and Paths
###################################################################################
if (( $# != 2 )) ; then
        echo -e "The script allows to filter proteome in fasta format from Ensembl with aims to keep the longest isoform per gene !"
      echo -e "The script needs 2 parameters: \n(1)The proteome input fasta file"
      echo -e "(2)The cleaned proteome output in fasta format"
      exit
fi

# Program heart:
 cat $1 | awk '/^>/ {if(N>0) printf("\n"); printf("%s\t",$1"\t"$4);N++;next;}{printf($0);} END {if(N>0) printf("\n");}' | awk -F '\t'  '{printf("%s\t%d\n",$0,length($3));}' | sort -t '	' -k2,2 -k4,4nr  | sort -t '	' -k2,2 -u -s | cut -f 1,2,3 | awk '{print $1"\t"$2"\n"$3}' | fold -w 60 > $2
```

BILS – Bioinformatics Infrastructure for Life Sciences

Correction: Remove isoform from proteome

Explanation of the script "remove isoforms":

1) `cat $1| awk '/^>/ {if(N>0) printf("\n"); printf("%s\t",$1"\t"$4);N++;next;} {printf($0);} END {if(N>0) printf("\n");}'` |\ #linearize fasta and print col1 col4 and seq linearized
2) `awk -F '\t' '{printf("%s\t%d\n",$0,length($3));}'` |\ #extract length on the 3th column
3) `sort -t '     ' -k2,2 -k4,4nr` |\#sort on column2, inverse length
4) `sort -t '     ' -k2,2 -u -s` |\# #sort on column 2, unique, stable sort (keep previous order)
5) `cut -f 1,2,3` |\ #cut 3column
6) `awk '{print $1"\t"$2"\n"$3}'` |\ # print col1 col2 col3 separated by tabulation
7) `fold -w 60` #pretty fasta = 60 letters per line

! Can also be written on command by line with pipe at the end of each line.

awk '/^>/ … <= every time there is the superior character
printf("%s",$0); => %s non useful

Awk and join command

Exercise from Matthew Webster Perl course:

http://blog.websterlab.eu/courses/perl/exercises/hashes_and_regular_expressions/
Exercice 3

http://blog.websterlab.eu/courses/perl/exercises/8-control-structures/
Exercise 2 from question 5

BILS – Bioinformatics Infrastructure for Life Sciences

Process control example:

```
# Launch training test
nbjobl=0
for (( i=$mini; i<=$maxi ; i=i+$interval)); do
        bsub -J trTe$i "augustus --gff3=on --species=$species$i $testFile$nbGeneRef | tee
$traingFile${i}Test.out > started.trte$i"
        ((nbjobl=nbjobl+1))
done
echo "nb launched jobs = $nbjobl"

sleep 5
nbjobs=1
while [ $nbjobs != 0 ]; do
      nbjobs=$(bjobs | grep -c "trTe")
      echo "nb training test jobs running= $nbjobs"
      sleep 30
done
```

In the for loop it's possible to launch a determined number of job. And check the number of job running each Xsecondes. If number job running inferior to nuber job authorized, launch a new job.

6) Downsampling a fastq

How get a random sample of a dataset ?

Correction:

```
paste f1.fastq f2.fastq |\ #merge the two fastqs
awk '{ printf("%s",$0); n++; if(n%4==0) { printf("\n");} else { printf("\t\t");} }' |\ #merge by group of 4 lines
shuf  |\ #shuffle
head |\ #only 10 records
sed 's/\t\t/\n/g' |\ #restore the delimiters
awk '{print $1 > "file1.fastq"; print $2 > "file2.fatsq"}' #split in two files.
```