Parsing in Unix egrep, sed, awk & regex

Patrick Fuchs

Outline

1. **Parsing**
2. **Regular expressions**
3. **egrep: line by line**
4. **sed: stream editor → very fast**
5. **Stream redirection**
6. **awk: very powerful → programming language**

Parsing

Definition: syntactic analysis

= mining data in text file(s) using

→ Unix tools: **`egrep`**, **`sed`**, **`awk`**, etc.

→ with a specific grammar: **regular expression** (*regex*)

Very powerful when used with Unix pipes (`|`)

Also usable in scripting languages (**python**, **perl**, **R**, etc)

Basic grep

General syntax: `grep expression filename`

Read `filename` (=text file) line by line, and print the current line if it contains `expression`

Example: `grep "ATOM" 1MSE.pdb`

→ output all lines containing `ATOM`

Basic grep (2)

- grep on multiple files: `grep expression file1 file2 file3`

```
[fuchs@rome OEB2012]$ grep LOCUS *.gbk
NC_001133.gbk:LOCUS       NC_001133  230208 bp    DNA     linear   PLN 16-JUN-2008
NC_001806.gbk:LOCUS       NC_001806  152261 bp    DNA     linear   VRL 24-AUG-2010
[fuchs@rome OEB2012]$
```

- Good practice: always use quotes!

  `grep ACCESSION X92210 file.gbk`

  ≠ `grep "ACCESSION X92210" file.gbk`

- Useful options:
  - `-l` : return file name only that match `expression`
  - `-v` : return lines that don't match `expression`
  - `-n` : return line numbers where `expression` matches
  - `-i` : case insensitive

Combine grep and find

Search recursively all files containing a word:

```
find . –name "*.gbk" –exec grep –l Drosophila {} \;
```

Regular expressions

Regular expression (regex) = filter strings with a specific grammar

`egrep`, `sed`, `awk` handle natively regex

```
egrep "^ATOM" 1MSE.pdb
```

→ `^ATOM` is a regex

`Python`, `Perl`, `R` also have libraries dedicated to regex

Syntax of regular expressions

regex are made of:

- **normal characters** (interpreted normally)

- **metacharacters** = special characters which means something for the parsing program

Example:

```
egrep "^ATOM" 1MSE.pdb
```

→ `^` means beginning of line

→ `ATOM` means `ATOM`

➔ print to screen all lines starting with the word `ATOM`

Syntax of regular expressions

Metacharacters

^	beginning of line
$	end of line
.	any character
[ABC]	A or B or C
(AB|AC|AZ)	AB or AC or AZ
[A-Z]	any upper case letter
[a-z]	any lower case letter
[0-9]	any numeral character
[^AB]	any character except A or B
****	escape the next metacharacter

```
egrep '(ALA|GLY)' 1MSE.pdb
```

→ print to screen all lines containing `ALA` or `GLY`

Syntax of regular expressions (2)

Metacharacters (counting)

`*`	0 or n times
`+`	1 time or more
`?`	0 or 1 time
`{n}`	n times
`{n,m}`	n to m times
`{n,}`	at least n times

Applies to the previous character or to the expression between ()

```
egrep 'A{3,}' NC_001133.gbk
```

→ print any line containing at least 3 consecutive `A`

`egrep` = extended grep

General syntax: `grep 'regex' filename`

Read `filename` (=text file) line by line, and print the current line if it matches `regex`
(same as grep but able to interpret regex; `grep -E` is equivalent to `egrep`)

Example:

```
egrep '(ALA|GLY)' 1MSE.pdb
```

→ print to screen all lines containing `ALA` or `GLY`

good pratice: Always use **single** quotes

```
egrep 'tgttagtgtt$' NC_001133.gbk
≠ egrep "tgttagtgtt$" NC_001133.gbk
```

`egrep` = extended grep

General syntax: `grep 'regex' filename`

Read `filename` (=text file) line by line, and print the current line if it matches the `regex`
(same as grep but able to interpret regex)

Example:
`egrep '(ALA|GLY)' 1MSE.pdb`
→ print to screen all lines containing `ALA` or `GLY`

GOOD ☺!

WRONG ☹!

good pratice: always use **single** quotes

```
egrep 'tgttagtgtt$' NC_001133.gbk
≠ egrep "tgttagtgtt$" NC_001133.gbk
```

Powerful example: get DNA sequence from a gbk file using `egrep`

```
egrep '^ +[0-9]+ [atgc ]+$' file.gbk
```

→ if it doesn't work, remove before the Windows carriage returns:

```
tr -d '\r' < file.gbk > file_OK.gbk
```

and rerun:

```
egrep '^ +[0-9]+ [atgc ]+$' file_OK.gbk
```

`sed` program

`sed` (***s****tream* ***ed****itor, son of* `ed`)

non interactive editor: sed reads a file (or stream) line by line and eventually does an action on the line

very powerful, very fast

General syntax: `sed [options] 'command' filename`

Good practice:

- option **`-r`** to have full regex (as in `egrep`)
- always use **single** quotes

`sed`: substitution

General syntax: `sed -r 's/regex/repl/g' filename`

`regex` is in indicated between `//`

`s` is a sed command= substitute
`g` is a sed command= global (substitute all occurrences of `regex` in the line)

Example:
`sed -r 's/^foo/fee/g' file`

→ replace all occurrences of `^foo` by `fee` in `file` and print output to screen

`sed`: multiple substitutions

General syntax: Use option `-e`

```
sed -r -e 's/regex1/repl1/g' -e \
's/regex2/repl2/g' filename
```

equivalent to

```
sed -re 's/regex1/repl1/g' -e  \
's/regex2/repl2/g' filename
```

Possible to combine any number of substitutions

Example:

```
sed -re 's/^atg//g' -e 's/ggg/ttt/g' file
```

→ replace all occurrences of `^atg` by nothing, then replace all occurrences of `ggg` by `ttt` in `file` and print output to screen

`sed`: same substitution on multiple files

General syntax:

```
sed –r 's/regex/repl/g' file1 file2 file3
```

Example:

```
sed -r 's/^LOCUS/JOKE/g' *.gbk
```

→ replace all occurrences of `^LOCUS` by `HELLO` in every `gbk file` and print all the output to screen

`vi`: substitution

Same syntax as `sed` ☺

While typing text in `vi`, if one wants to do an automatic substitution:

```
Esc
:%s/regex/repl/g
```

`sed`: partial printing (1)

General syntax:

```
sed -rn 'EXPRp' filename
sed -rn 'EXPR1,EXPR2p' filename
```

`-n`: activates partial printing
`p` is a sed command= print
`EXPR`, `EXPR1` & `EXPR2` can be a line number or a regex between `//`

Examples:

```
sed -rn '/regex1/,/regex2/p' file
```

→ print all lines of `file` starting from 1st occurrence of `regex1` up to 1st occurrence of `regex2`

`sed`: partial printing (2)

```
sed –rn '/regex1/p' file
```

→ print all lines of file matching regex1 (same as egrep)

```
sed –rn '1,10p' file
```

→ print lines 1 to 10 to screen

```
sed –rn '15p' file
```

→ print line 15 to screen

```
sed –rn '100,$p' file
```

→ print lines 100 to last line to screen

Beware: here, `$` = last line ≠ `$` in a regex

`sed`: partial printing (3)

```
sed –rn '/regex1/,100p' file
```

→ print all lines from the 1st occurrence of regex1 up to line 100

```
sed –rn '10,/regex1/p' file
```

→ print all lines from line 10 to the 1st occurrence of regex1

`sed`: deleting lines

General syntax:

```
sed -r 'EXPRd' filename
sed -r 'EXPR1,EXPR2d' filename
```

`d` is a sed command = delete

`EXPR`, `EXPR1` & `EXPR2` = line number or regex between `//`

Examples:

```
sed -r '/regex1/d' file
```

→ delete all lines matching `regex1` and print output to screen

(*i.e.* print all lines except those matching `regex1`, `file` is not modified)

`sed`: deleting lines (2)

```
sed '100d' file
```

→ delete line 100 and print output to screen

```
sed '10,20d' file
```

→ delete lines 10 to 20 and print output to screen

`sed`: quitting

General syntax:

```
sed -r 'EXPRq' filename
```

`q` is a sed command= quit

`EXPR` is a line number or a regex between `//`

Examples:

```
sed '100q' file
```

→ print all lines up to line 100

```
sed -r '/regex/q' file
```

→ print all lines up to 1st occurrence of `regex`

Beware: don't use a double address with command `q`

`sed`: transliterate

General syntax:

```
sed 'y/source/dest/' filename
```

`y` is a sed command= transliterate

Transliterate the characters which appear in source to the corresponding character in dest

Examples:

```
sed 'y/atgc/tacg/' file
```

→ replace `a` by `t`, `t` by `a`, `g` by `c` and `c` by `g`

`sed`: modify input file directly

Option `-i`: modifies directly the filtered file (no output on `stdout`)

Examples:

```
sed -ri 's/foo/fee/g' file
```

→ replaces all occurrences of `foo` by `fee` in `file` (**beware**, `file` is modified!)

```
sed -rni '/regex1/,/regex2/p' file
```

→ prints all lines of `file` starting from 1st occurrence of `regex1` until 1st occurrence of `regex2` (**beware**, `file` is modified!)

`sed`: modify many input files

Option `-s`: combined with `-i`, modifies directly each file (no output on `stdout`)

Examples:

```
sed -ris 's/LOCUS/JOKE/g' *.gbk
```

→ replaces all occurrences of `LOCUS` by `JOKE` in each `gbk file` (**beware**, each `gbk file` is modified!)

Last recommendations

1) During substitution, **regex are greedy**!
Beware with `+` and `*`

Search '`ATG+`' in '`AAATCCTAATATGGGTA`'

replaces '`AAATCCTAATGGGTA`'

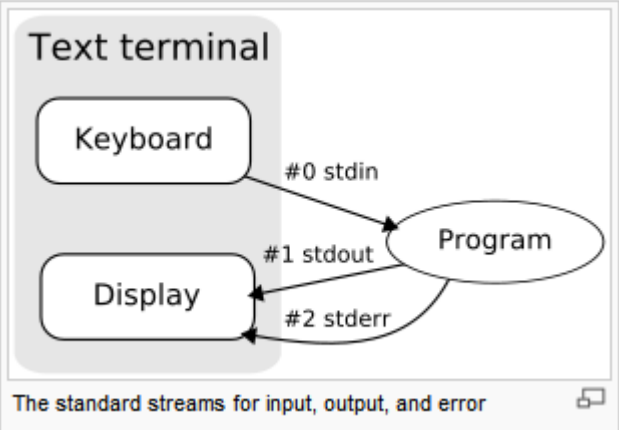
although '`AAATCCTAATGGGTA`' or '`AAATCCTAATGGGTA`'
both match the regex!

2) Beware of ambiguous regex with `+` and `*`!

Unix streams

Unix streams refer to instructions going into or flowing out of processes

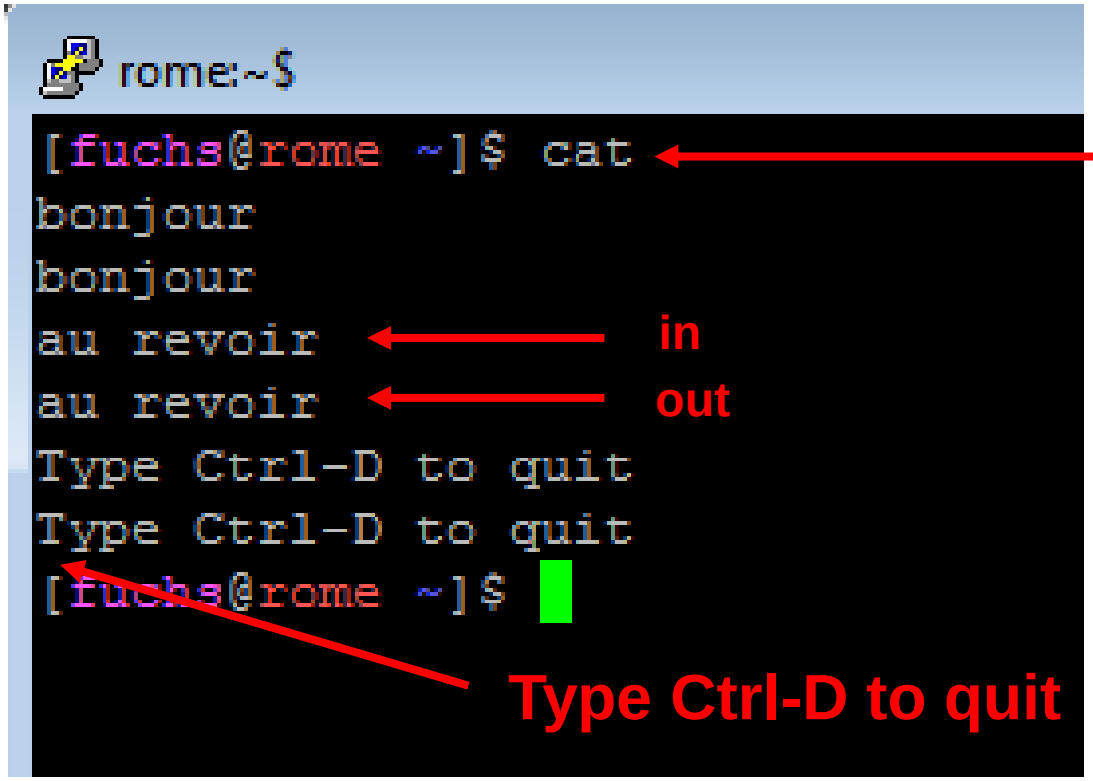
- `stdin`: standard input (keyboard)
- `stdout`: standard output (screen)
- `stderr`: standard error (screen)



http://en.wikipedia.org/wiki/Redirection_(computing)

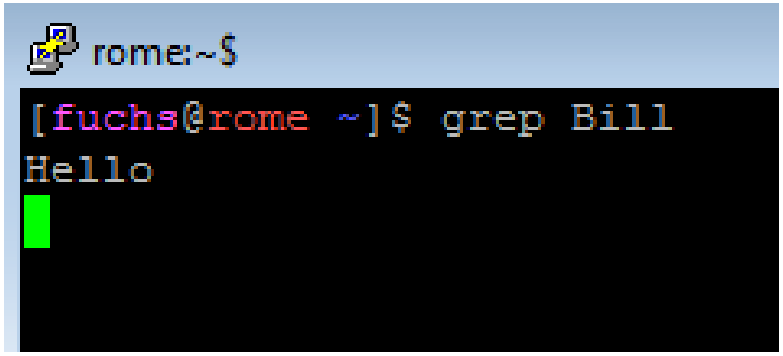
What does this mean?

Many Unix commands read `stdin` (keyboard) and give output to `stdout` (screen) by default

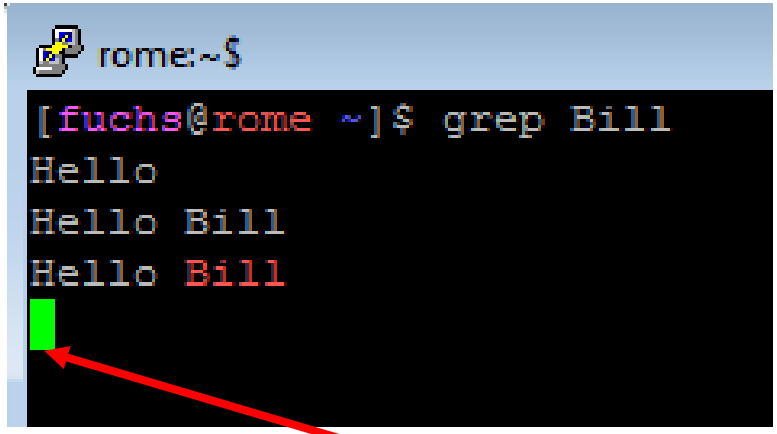


`cat` invoked without argument reads `stdin` and gives output to `stdout`

Another example



Why is there no output?



Why is there output?

Type Ctrl-D to quit

Stream redirection

`<` `stdin` redirection from a file
`>` `stdout` redirection to a file
`>>` `stdout` redirection at the end of a file

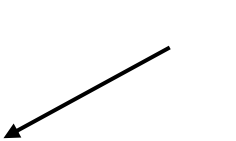
Examples:

`ls -l /etc > toto` (beware if toto exists!)

`ls -l /bin >> toto` (beware if toto exists!)

`cat < toto`

Same output?




```
cat < toto
```

Yes!

…but, formerly different

```
cat toto
```

`cat` feature

So is `stdin` redirection useful?

Yes! Some unix programs read standard input only

Example:

```
tr -d '\r' < file
```

So is `cat` useful?

Yes! Useful for concatenation:

```
cat file1 file2 file3
cat file1 file2 > output
```

Unix developers are fun, try that one:

```
tac file
```

Reverse a string

`rev` ➔ reverse lines of file(s) or stream

Examples:

```
[fuchs@rome IJM_sed_awk]$ more toto
Want chocolate
me too!
[fuchs@rome IJM_sed_awk]$ rev toto
etalocohc tnaW
!oot em
[fuchs@rome IJM_sed_awk]$ echo \
"I love chocolate" | rev
etalocohc evol I
```

Using `>` and `>>` safely

`set -o noclobber`

→ Prevents '`ls > file`' to overwrite `file` if it already exists

Save it to your `~/.bashrc` to have it always active

Filter commands

`head file`: 10 first lines
`tail file`: 10 last lines
`grep expression file`: you already know ☺
`sort file`: alphabetical sorting (by default)
`wc file`: word counter

Examples

`sort –nk 2 file` → sort by numerical order the second column of `file`

`wc –l file` → give the number of lines of `file`

VERY useful pipes

`cmd1 | cmd2` ➔ connects `stdout` of `cmd1` to `stdin` of `cmd2`

Examples

```
ls -l | sort
ls -l /etc | grep csh
```

Remarks:

- **Never** provide a file name for the 2nd command!
- One can use as many pipes as wanted:

```
ls -l /etc | grep csh | sort
```

Powerful example

Parse a sequence from a gbk file and evaluate the reverse complement:

```
sed -n '/^ORI/,/^\/\//p' file.gbk | \
sed -r '/^(ORI|\/\/)/d' | \
sed -r 's/[0-9 ]//g' | \
sed -r ':a;N;$!ba;s/\n//g' | \
sed 'y/atgc/tacg/' | rev
```

Powerful example (2)

Same example using `tac` and `tr`

```
sed -n '/^ORI/,/^\/\//p' file.gbk | \
sed -r '/^(ORI|\/\/)/d' | \
sed -r 's/[0-9 ]//g' | \
sed 's/./&\n/g' | tac | \
tr -d '\n' | sed 'y/atgc/tacg/'
```

Other redirections

`cmd << flag` reads `stdin` until `flag`

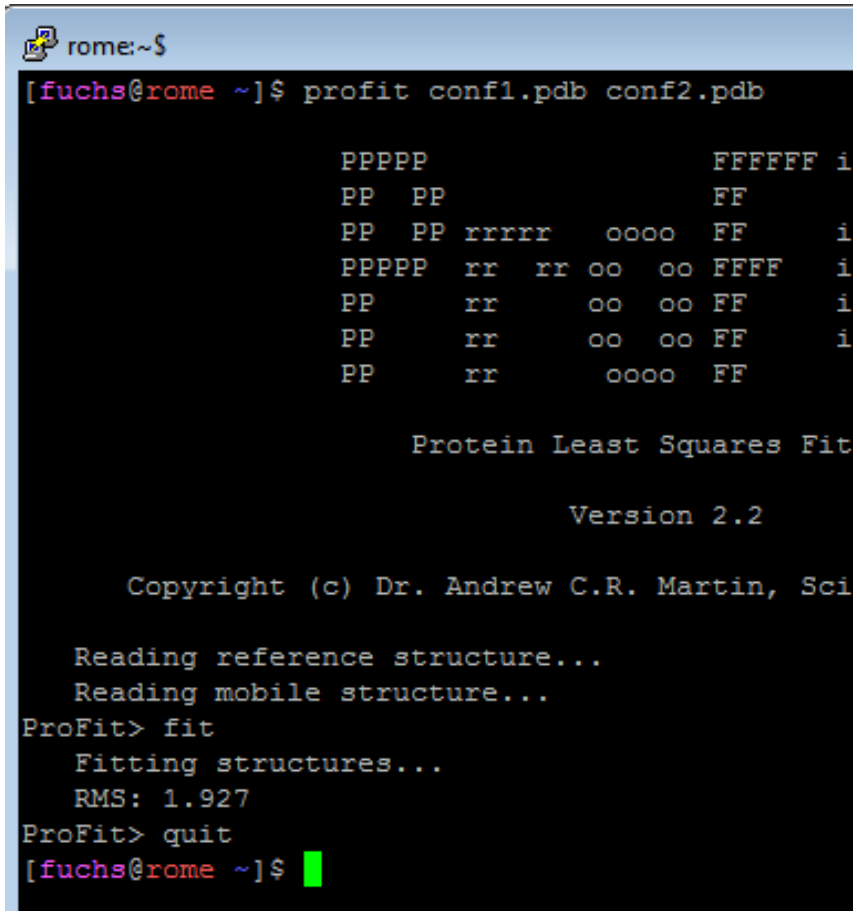
`cmd >& file` redirection of `stdout` and `stderr` to `file`

`cmd 1> file1 2> file2` redirection of `stdout` to `file1` and of `stderr` to `file2`

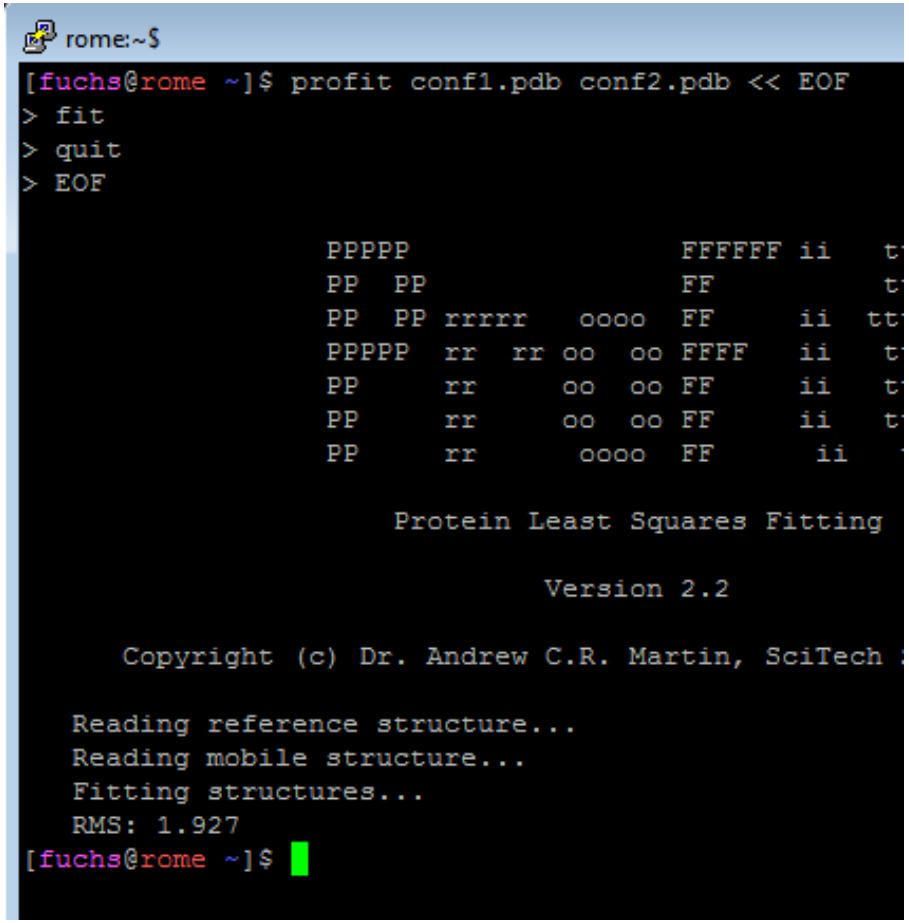
`/dev/null` = "black hole" of the computer (always empty)

Is << useful? Yes!

without <<



with <<

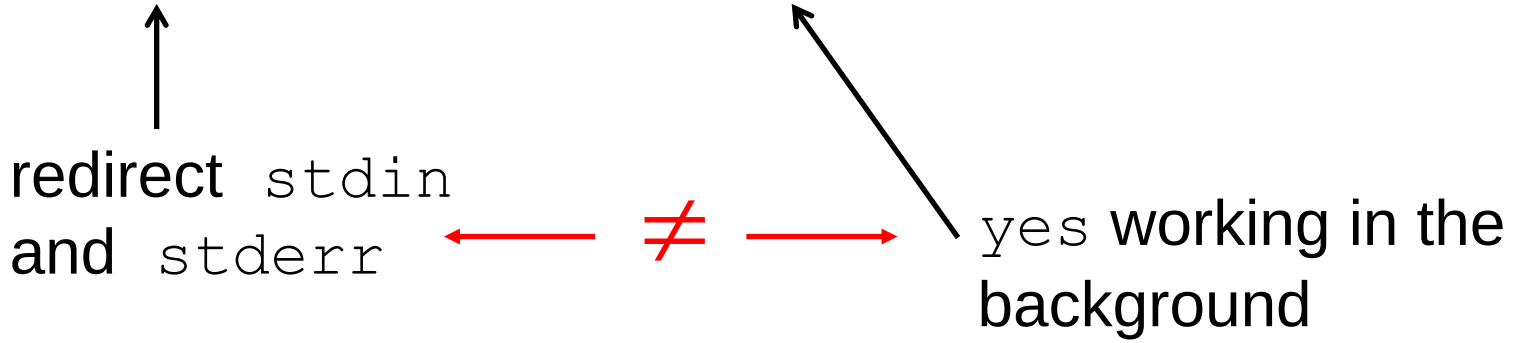


CPU heater

`yes` = print `y` indefinetly to the screen

CPU is working hard for nothing ☺

```
yes > /dev/null
yes >& /dev/null &
```



Next week

awk!

hint: named from authors (Aho, Weinberger, Kernighan)

Thanks for your attention!

`awk` program

Very powerful parsing program including classical features (variables, loops, tests…) = real programming language

In this course we'll study mainly `awk` with the command line… but possible to write scripts

Same concept as `egrep` and `sed`: `awk` reads a file or flow line by line and does an action if a test is true

General Syntax: `awk 'test{action(s)}' file`

With a pipe: `cmd1 | awk 'test{action(s)}'`

Basic example

Equivalent of `egrep`:

```
awk '$0 ~ /regex/{print $0}' file
```

`regex` indicated between `/  /` (like in sed)

`;` command separator within `{  }`

Good practice:

1) always use `awk --posix` (posix is a norm for regex, e.g. extended metacharacters suchs as `{}` are supported)

2) always use single quotes

Variables in `awk`

General variables:

```
var=1
var=3.14
var="toto"
```

Predifined variables: apply to the current line (= line beeing read by `awk`)

`$0` (whole) current line

`NR` line number

`FS` field separator (default: any combination of space(s) and/or tabulation(s))

`NF` number of fields (of current line)

`$x` field `x` (`x` runs from 1 to `NF`)

`FILENAME` (note the upper case)

Tests in awk

Recall: `awk 'test{command(s)}' file`

A test can be done

- on a numerical value
  - `==` equal to
  - `!=` not equal
  - `>` greater than
  - `<` lower than
  - `>=` greater or equal
  - `<=` lower or equal
- on a regex
  - `~` matches
  - `!~` doesn't match

Tests in awk (2)

Examples

```
awk '$0 ~ /^ORIGIN/ {print $0}' file
awk '$1 ~ /ORIGIN/ {print $0}' file
awk 'NR == 10 {print $0}' file
awk 'NR >= 10 {print $0}' file
awk '$0 !~ /ORIGIN/ {print $0}' file
```

Boolean operators: **&&** (and), **||** (or), **!** (not)

```
awk '$0 ~ /^ *[0-9]+/ && NF == 10 {print $0}' file
```

Actions in awk

- Actions always between `{ }`
- Examples of actions
  - text printing: `{print "bonjour",$2}`
    `{print "bonjour" $2}`
  - variable modification: `{var=var+1}`
  - variable definition: `{i=2}`
  - Combining actions: `{print $0 ; i=0}`
  - Use other functions
    - pre-defined awk functions
    - other tests, loops, etc…

comma adds a space between the arguments

no comma ➔ arguments concatenated

semi-column = command separator

Default behavior

- No test → every line is considered

```
awk '{print $1}' file
```

- Only a regex between `//` → whole line (`$0`) is tested for regex matching

`awk '/^ATOM/{print $0}' file` (equivalent to)

```
awk '$0 ~ /^ATOM/{print $0}' file
```

- No action → print (whole) current line (`{print $0}`)

`awk '$1 == /ATOM/' file` (equivalent to)

```
awk '$1 == /ATOM/ {print $0}' file
```

Print part of a file

`//,//` supported (like in sed)

```
awk '/^ORIGIN/,/\/\//' NC_001806.gbk
```

From line 5 to 10 (inclusive)

```
awk 'NR ==1,NR == 10' NC_001806.gbk
```

Some real examples

Get lines with coordinates from a pdb file

```
awk '/^ATOM/' 1MSE.pdb
```

Get Cα x,y,z coordinates from a pdb file

```
awk '/^ATOM/ && $3 == "CA" {print $7, $8, $9}' 1MSE.pdb
```

Get DNA sequence from a gbk file

```
awk '/^ +[0-9]+ [atgc ]+$/ {$1="" ; print $0}' \
file.gbk
```

Some real examples (2)

Get DNA sequence from a gbk file on a single line

```
awk '/^ +[0-9]+ [atgc ]+$/ \
{gsub(/[0-9 ]/,"",$0) ; \
printf "%s", $0}' file.gbk
```

multiple regex substitution

formatted printing

continue command on next line

Modify field separator

`-F` option allows changing field separator:

```
awk -F: '{print $1}' /etc/passwd
```

no space after `-F`

Use `\` when field separator might be interpreted by bash

```
awk -F\" '{print $2}' file
```

Also doable using `FS` variable within `awk` execution:

```
awk '/^ +gene +[0-9]/{sub(/ +gene +/,"",$0)\
  ; FS="." ; print $1,$3}' NC_001806.gbk
```

single regex substitution

Next week teaser

awk scripting!

Thanks for your attention!

awk scripting

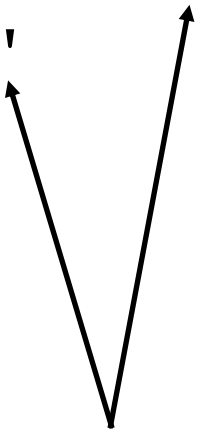
- awk scripting → use option `-f` :
  `awk -f myscript.awk file`
- Block of instructions defined between `{}`
- Instruction separator: `;` (same line) or new line
- Two special "areas":
  - before awk starts to read and process the 1st line: `BEGIN {instructions}`
  - after reading and processing of the last line: `END{instructions}`

A real example

- Calculate center of mass of a protein (Cα only):

```
awk 'BEGIN {x=0 ; y=0 ; z=0 ; count=0}
/^ATOM/ {count++ ; x+=$6 ; y+=$7 ; z+=$8}
END {print x/count,y/count,z/count}'
1BTA.pdb
```

Problem: the line gets very long!



\ not mandatory at the command line

Rewriting of the COM extractor

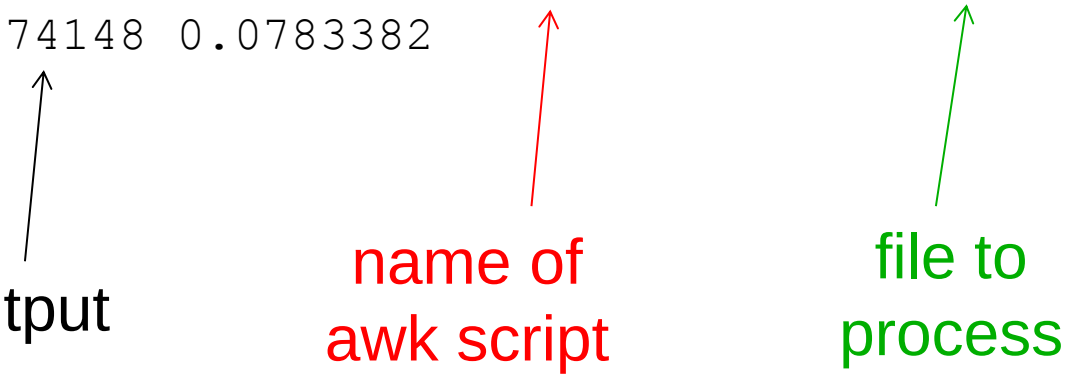
extract_CA_com.awk

```
BEGIN {
x=0 ; y=0 ; z=0
count=0
}

/^ATOM/ {
count++
x+=$6 ; y+=$7 ; z+=$8
}

END {
print x/count , y/count , z/count
}
```

More readable ☺

Run Ca COM extractor

```
[fuchs@rome ~]$ awk -f extract_CA_COM.awk 1BTA.pdb
0.109569 0.174148 0.0783382
```




```
[fuchs@rome ~]$ cat 1BTA.pdb | awk -f extract_CA_COM.awk
0.109569 0.174148 0.0783382
```

also usable with pipes!