Plateforme Bioinfo Occitanie Toulouse

Genotoul Bioinfo

Training Day

Modify and extract information from large text files with

sed & awk

Plateforme Bioinfo Occitanie Toulouse

GENOTOUL Bioinfo

Objectives

At the end of the day, you will be able to use command line in order to :

- Edit big files (beyond excel files)
- Extract information from big files
- Build new files from existing ones

Plateforme Bioinfo Occitanie Toulouse

Genotoul Bioinfo

Planning of the day

Part I : 09h00 - 10h30

Regular expressions - TP1

Part II : 10h45 - 12h30

File editing with `sed` - TP2

Part III : 14h00 - 17h00

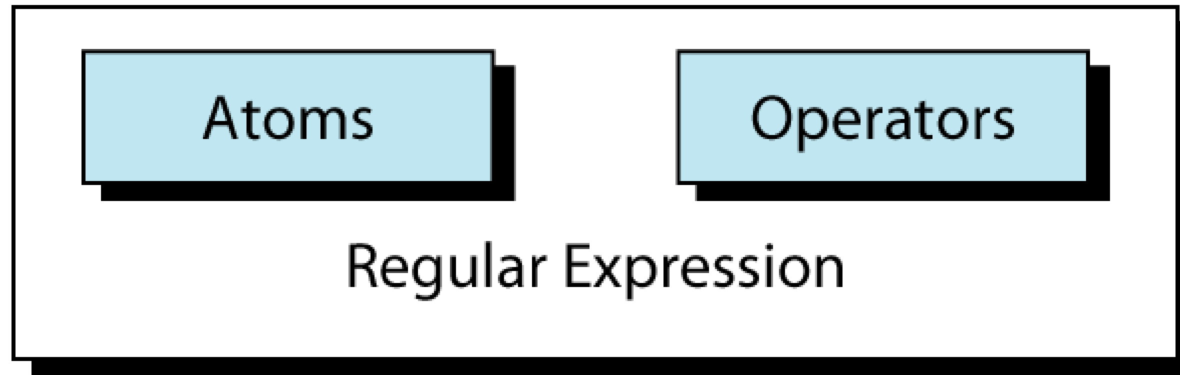
Extracting information with `awk` - Combining unix `sed` and `awk` within pipes - TP3

Part I
Regular expressions

Regular expression ?

- Commonly called regex
- A simple set of characters with special meanings (called metacharacters) to test for matches quickly and easily.
- Regular expressions are used thoughout UNIX:
  - grep
  - sed
  - awk
  - ...

Regular expression ?



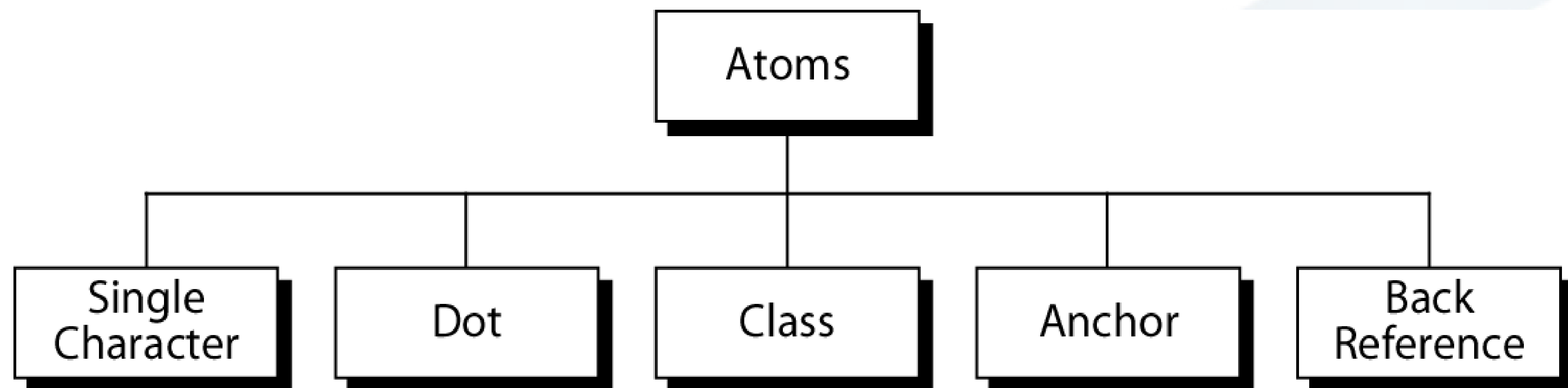
An atom specifies what text is to be matched and where it is to be found.

An operator combines regular expression atoms.

Regular expression

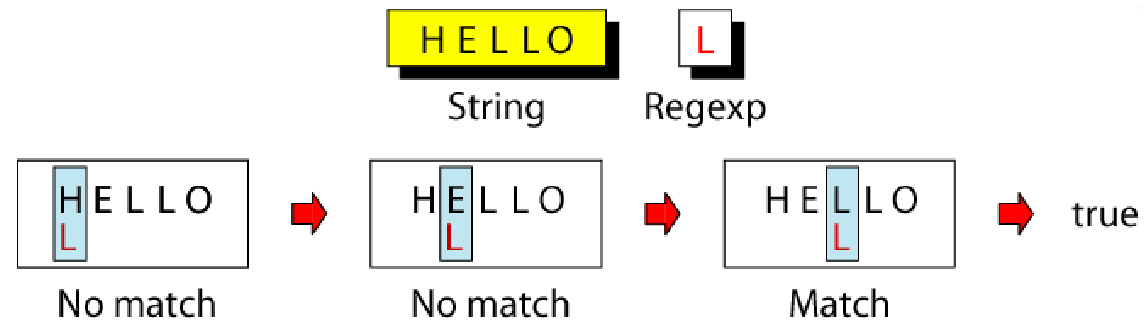
Atoms

An atom specifies what text is to be matched and where it is to be found.

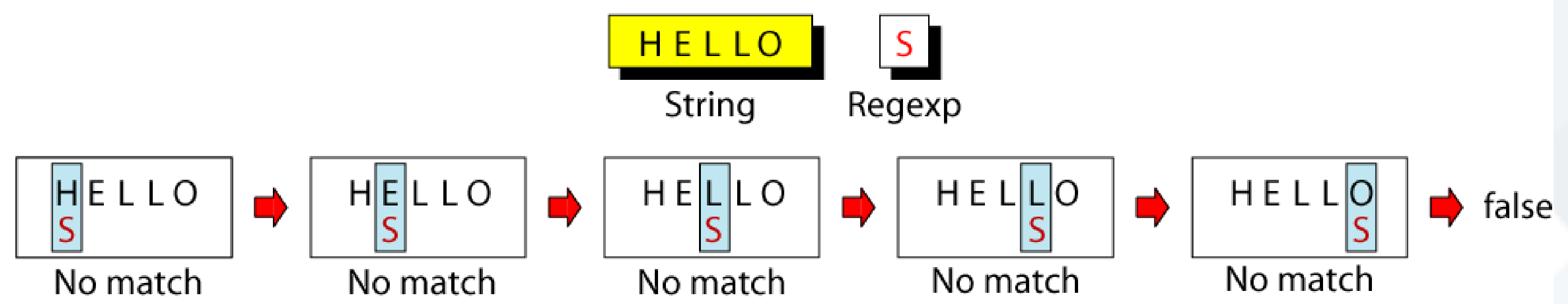


Single-Character Atom

A single character matches itself



(a) Successful Pattern Match



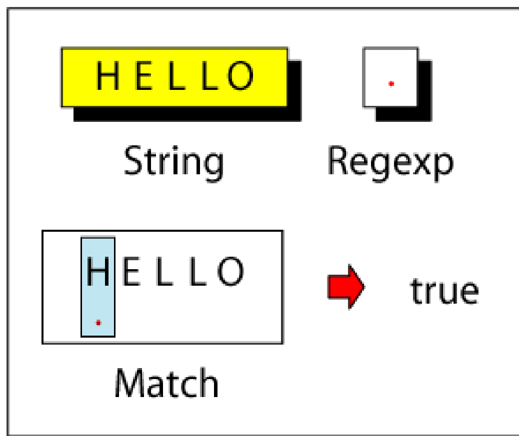
(b) Unsuccessful Pattern Match

Regular expression

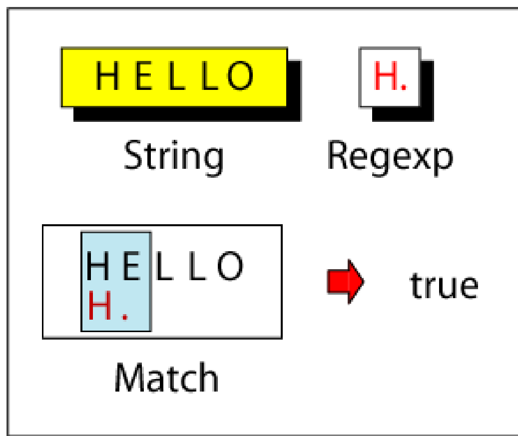
Atoms

Dot Atom : .

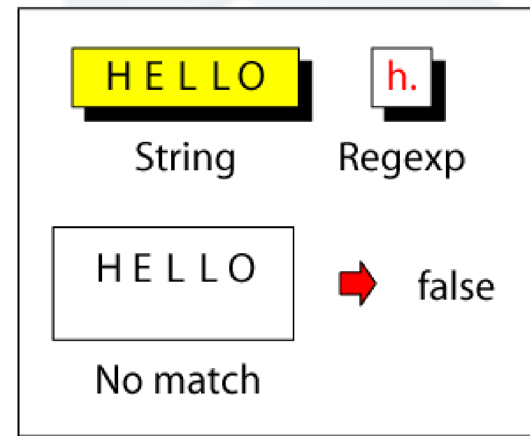
matches any single character except for a new line character (\n)



(a) Single-Character



(b) Combination–True

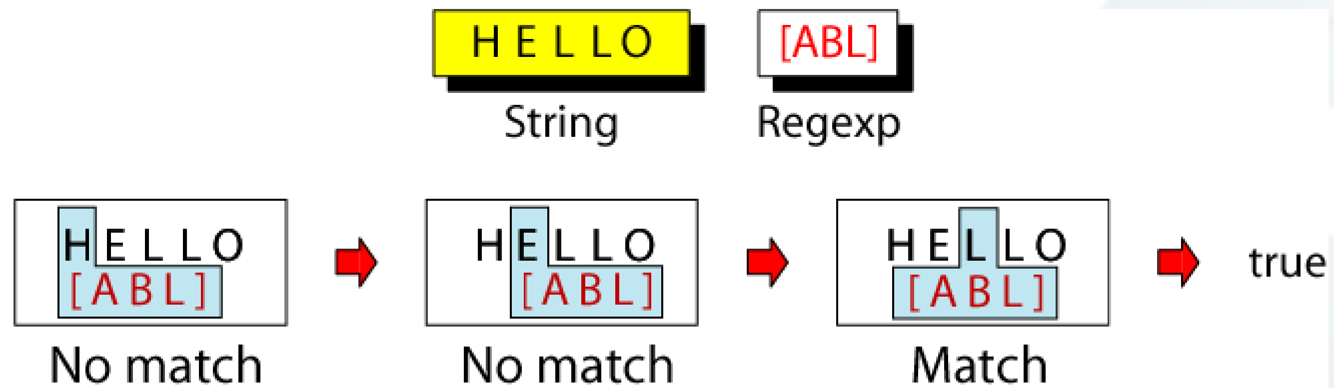


(c) Combination–False

Class Atom

matches only single character that can be any of the characters defined in a set:

Example: [ABC] matches either A, B, or C.



Notes:

1) A range of characters is indicated by a dash, e.g. [A-Q]
2) Can specify characters to be excluded from the set, e.g. [^0-9] matches any character other than a number.

Regular expression

Atoms

Example: Classes

RegExpr	Means	RegExpr	Means
[A-H]	[ABCDEFGH]	[^AB]	Any character except A or B
[A-Z]	Any uppercase alphabetic	[A-Za-z]	Any alphabetic
[0-9]	Any digit	[^0-9]	Any character except a digit
[[a]	[or a	[]a]	] or a
[0-9\-]	digit or hyphen	[^\^]	Anything except^

Anchors

Anchors tell where the next character in the pattern must be located in the text data.

Anchor	Means	Example
^	Beginning of line	One line of text.\n
$	End of line	One line of text.\n
\<	Beginning of word	One line of text.\n
\>	End of word	One line of text.\n

Regular expression
Atoms

Back References: \N

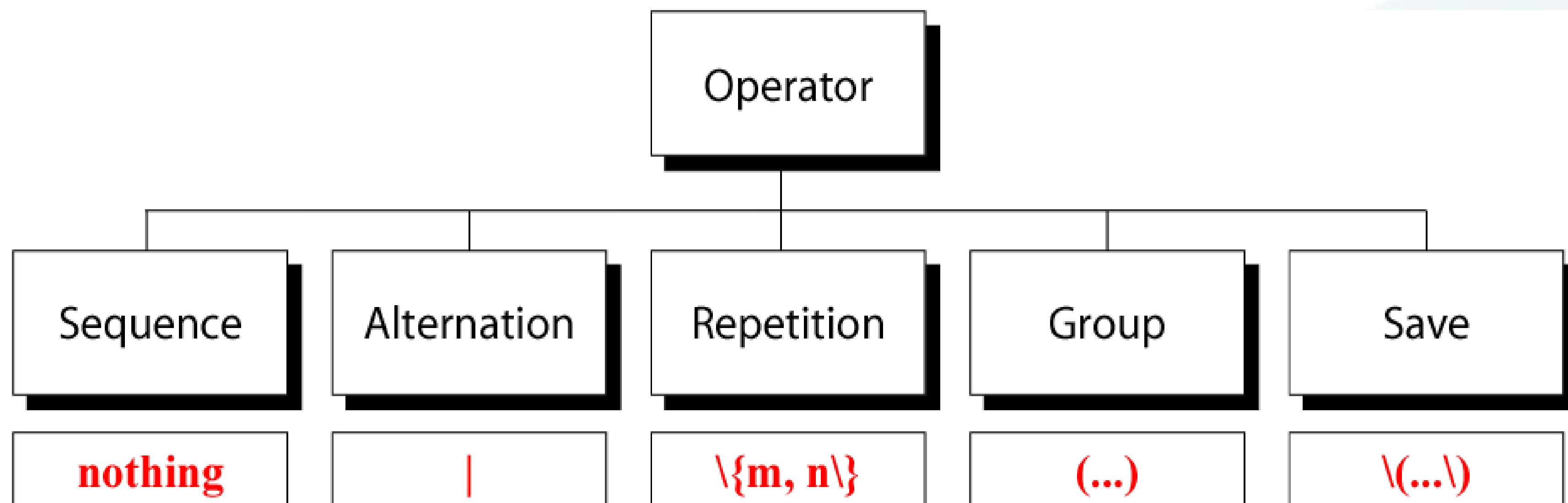
- used to retrieve saved text in one of **nine** buffers
- can refer to the text in a saved buffer by using a back reference:

  ex.: \1 \2 \3 ...\9

- more details on this later

Regular expression

Operators



Regular expression

Operators

Sequence Operator

In a sequence operator, if a series of atoms are shown in a regular expression, there is no operator between them.

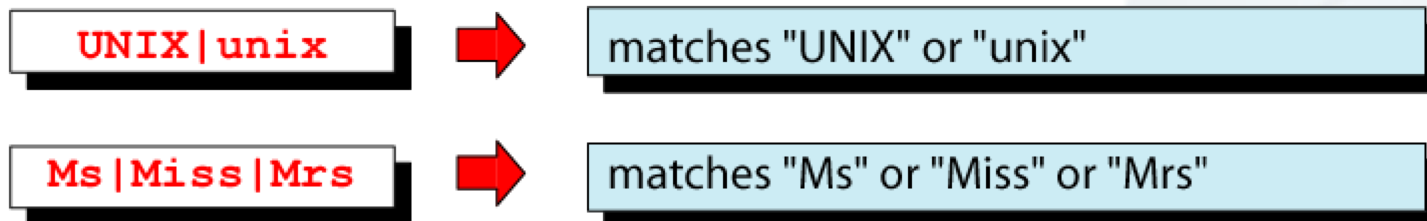
Pattern	Meaning
dog	matches the pattern "dog"
a..b	matches "a" , any two characters, and "b"
[2-4][0-9]	matches a number between 20 and 49
[0-9][0-9]	matches any two digits
^$	matches a blank line
^.$	matches a one-character line
[0-9]-[0-9]	matches two digits separated by a "-"

Regular expression

Operators

Alternation Operator: | or \|

operator (| or \|) is used to define one (depends on version of “grep”) **or** more alternatives

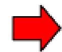


Repetition Operator: \{...\}

The repetition operator specifies that the atom or expression immediately before the repetition may be repeated.

```
\{m , n\}
```

matches previous character m to n times.

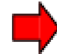
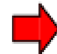
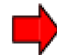
`A\{3 , 5\}`		matches "AAA" , "AAAA", or "AAAAA"
`BA\{3 , 5\}`		matches "BAAA" , "BAAAA", or "BAAAAA"

Regular expression

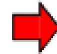
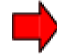
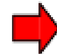
Operators

Basic Repetition Forms

Formats

\{m\}		matches previous atom exactly m times
\{m, \}		matches previous atom m times or more
\{, n\}		matches previous atom n times or less

Examples

CA\{5\}		CAAAAA
CA\{3,\}		CAAA, CAAAA, CAAAAA, …
CA\{,2\}		C, CA, CAA

Regular expression

Operators

Short Form Repetition Operators:

Formats

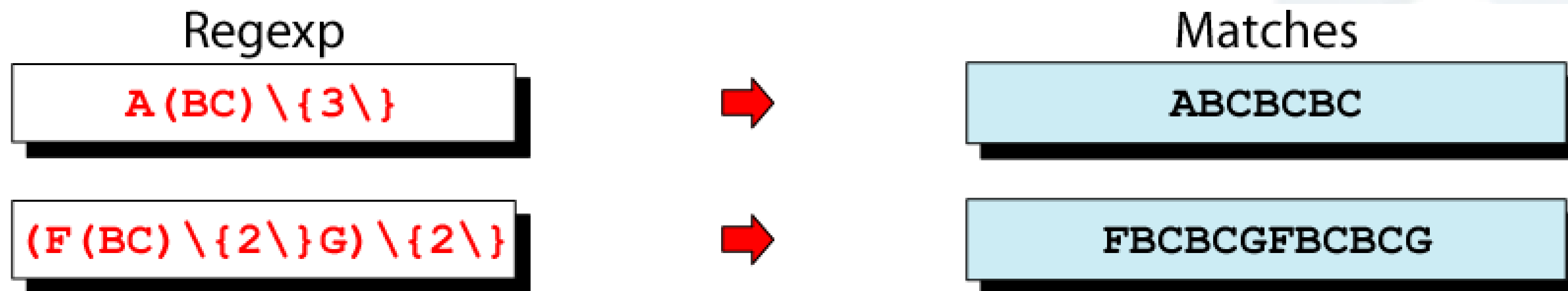
Format	Meaning
*	special case: matches previous atom zero or more times
+	special case: matches previous atom one or more times
?	special case: matches previous atom 0 or one time only

Examples

Example	Matches
BA*	B, BA, BAA, BAAA, BAAAA, . . .
B.*	B, BA . . . BZ, BAA . . . BZZ, BAAA . . . BZZZ, . . .
.*	zero or more characters
.+	one or more characters
[0-9]?	zero or one digit

Group Operator

In the group operator, when a group of characters is enclosed in parentheses, the next operator applies to the whole group, not only the previous characters.



Note: depends on version of “grep”
use \(and \) instead

Genotoul Bioinfo

Regular expression
Examples

- Pipe the output of the « ls –l » command to grep and list/select only directory entries

```
ls -l | grep '^d'
```

- Display the number of lines where the pattern was found

```
ls -l | grep -c '^d'
```

- Look at the file `/home/cgaspin/TP/TP_REGEX/All_RFAM_bostaurus_ncRNA.gff`.
  - Print all the line containing the word « SNORA5 » .

```
cd /home/cgaspin/TP/TP_REGEX/
grep '\<SNORA5\>' All_RFAM_bostaurus_ncRNA.gff
```

  - Print the lines that containing either the expression “SRP” or the expression « MRP »

```
grep 'SRP\|MRP' All_RFAM_bostaurus_ncRNA.gff
```

  - Print all lines containing « _ » followed by either « MRP » or « SRP ».

```
grep '_\(SRP\|MRP\)' All_RFAM_bostaurus_ncRNA.gff
```

  - Print all the lines corresponding to chromosome 1 (beginning by « 1 »)

```
grep '^\<1\>' All_RFAM_bostaurus_ncRNA.gff
```

Regular expressions
Metacharacters

- . : un caractère quelconque
- ^ : début de ligne
- $: fin de ligne
- \< : début d'un mot
- \> : fin d'un mot
- \ : considère littéralement le caractère suivant
- - : spécifie un intervalle (A-Z)
- X* : zéro ou plus d'occurrences de X
- X+ : une ou plus occurrences de X
- X?: zéro ou une occurrence unique du caractère X
- […] : plage de caractères permis
- [^…] : plage de caractères interdits
- \{n\} : pour définir le nombre de répétitions n du caractère placé devant
- \{n,m\} : **n** to **m** of preceding item (plus others)
- \(...\) : désigne une sous-chaîne
- \entier : désigne le numéro de sous-chaîne

Part I

TP1

- Exercises

Part II
File editing with `sed`

Introduction to sed

Sed: Stream-oriented, Non-Interactive, Text Editor

- **Look** for patterns one line at a time
- **Change** lines of the file
- **Non-interactive text editor**
  - Editing commands can come in as *script*
  - There is an interactive editor *ed* which accepts the same commands
- A Unix **filter**
  - Superset of previously mentioned tools

Syntax

sed [-n] [-e] [‘command’] [file…]
sed [-n] [-f scriptfile] [file…]

Options

- “-n” : only print lines specified with the print command
- “-f” scriptfile : next argument is a filename containing editing commands
- “-e” command : the next argument is an editing command rather than a filename, useful if multiple commands are Specified

Brackets **{}** can be used to apply multiple commands to an address

```
[/pattern/[,/pattern/]]{
command1
command2
command3
}
```

`sed` command line

- A `sed` command line consists of up to two ***addresses*** and an ***action***, where the ***address*** can be a regular expression or a line number.
- If the addresses are not given the action is performed on all the lines.

```
>echo day
day
>echo day | sed -e 's/day/night/'
night
```

```
>echo "day" > day.txt
>sed -e 's/day/night/' < day.txt
night
>sed -e 's/day/night/' day.txt
night
```

sed commands

- General form :
  - *[address[, address]][!]command [arguments]*

- *sed* copies each input line into a *pattern space :*
  - if the address of the command matches the line in the *pattern space*, the command is applied to that line,
  - if the command has no address, it is applied to each line as it enters *pattern space,*
  - if a command changes the line in *pattern space*, subsequent commands operate on the modified line.

- When all commands have been read, the line in *pattern space* is written to standard output and a new line is read into *pattern space.*

sed commands
address

- An **address** can be either
  - a line number or
  - a pattern

- An address is enclosed in slashes (*/pattern/*).
  - A pattern is described using *regular expressions* (Basic Regular Expressions, as in **grep**)
  - If no pattern is specified, the command will be applied to **all** lines of the input file.
  - To refer to the last line: **$**

sed commands

address

- Most commands will accept two addresses :
  - if only one address is given, the command operates only on that line,
  - if two comma separated addresses are given, then the command operates on a range of lines between the first and second address, inclusively.

- The **!** operator can be used to negate an address, ie; *address!command* causes *command* to be applied to all lines that do ***not*** match *address.*

- The **I** operator can be used after the address to ignore character case.

Plateforme Bioinfo Occitanie Toulouse

Genotoul Bioinfo

"s" substitution function

- Substitute the first or all occurrences of a string by another one
  - `sed "s/toto/TOTO/" file`
    → substitute the first occurrence of « toto » by « TOTO »
  - `sed "s/toto/TOTO/3" file`
    → substitute the 3rd occurrence of « toto » by « TOTO »
  - `sed "s/toto/TOTO/g" fichier`
    → substitute all occurences of « toto » by « TOTO »
  - `sed "s/toto/TOTO/p" fichier`
    → print the lines with subtitution done
  - `sed "s/toto/TOTO/w resultat" fichier`
    → print the lines with subtitution done in a file named `resultat`
  - `sed -e "s/[Ff]raise/FRAISE/g" fichier`
    → substitute all occurences of « Fraise » or « fraise » by « FRAISE »
  - `sed "s/\(.*\)-\(.*\)/\2-\1/" fichier`
    → permute the two fields separated by a « - »

/black/!s/cow/horse/ would substitute "horse" for "cow" on all lines except those that contained "black"

Hands-on sed substitution

```
>cat inputfile.txt
one two three, one two three
four three two one
one hundred
>sed 's/one/ONE/' < inputfile.txt
ONE two three, one two three
four three two ONE
ONE hundred
```

What is remarkable in the example presented here above ?

```
>sed 's/one/ONE/2' < inputfile.txt
one two three, ONE two three
four three two one
one hundred
>sed 's/one/ONE/g' < inputfile.txt
ONE two three, ONE two three
four three two ONE
ONE hundred
```

Hands-on sed substitution

```
>cat inputfile2.txt
one two three, one two three, one two three, one two three
four three two one, four three two one
one hundred, one hundred
>sed -e '1s/one/ONE/g' < inputfile2.txt
>sed -e '/hundred/s/one/ONE/g' < inputfile2.txt
>sed -e '/^one/s/one/ONE/g' < inputfile2.txt
```

What are the results of the three command lines ?

```
>sed -e '1s/one/ONE/g' < inputfile2.txt
ONE two three, ONE two three, ONE two three, ONE two three
four three two one, four three two one
one hundred, one hundred
>sed -e '/hundred/s/one/ONE/g' < inputfile2.txt
one two three, one two three, one two three, one two three
four three two one, four three two one
ONE hundred, ONE hundred
>sed -e '/^one/s/one/ONE/g' < inputfile2.txt
ONE two three, ONE two three, ONE two three, ONE two three
four three two one, four three two one
ONE hundred, ONE hundred
```

Genotoul Bioinfo

Hands-on sed substitution

What is the result of the command line ?

```
>cat inputfile3.txt
one two three, one two three, one two three, one two three
four three two one, four three two one
one hundred, one hundred
one two three, one two three, one two three, one two three
four three two one, four three two one
one hundred, one hundred
>sed -e '2,3s/one/ONE/g' < inputfile3.txt
```

Hands-on sed substitution

Build a command line to put all

- 'two' word in uppercase for lines beginning with 'one' and
- 'three' word in uppercase for lines having numbers between 3 and 5.

```
[cklopp@beraldi:/tmp] cat inputfile3.txt
one two three, one two three, one two three, one two three
four three two one, four three two one
one hundred, one hundred
one two three, one two three, one two three, one two three
four three two one, four three two one
one hundred, one hundred
[cklopp@beraldi:/tmp] sed -e '/^one/{s/two/TWO/g;s/three/THREE/g}' inputfile3.txt
one TWO THREE, one TWO THREE, one TWO THREE, one TWO THREE
four three two one, four three two one
one hundred, one hundred
one TWO THREE, one TWO THREE, one TWO THREE, one TWO THREE
four three two one, four three two one
one hundred, one hundred
```

or

```
>sed -e '/^one/s/two/TWO/g' < inputfile3.txt | sed -e '3,5s/three/THREE/g'
one TWO three, one TWO three, one TWO three, one TWO three
four three two one, four three two one
one hundred, one hundred
one TWO THREE, one TWO THREE, one TWO THREE, one TWO THREE
four THREE two one, four THREE two one
one hundred, one hundred
```

"d" deletion function

- **sed '[address1][,address2]d' file**
  - Delete the addressed line(s) from the pattern space; line(s) not passed to standard output.
  - A new line of input is read and editing resumes with the first command of the script.

"d" deletion function

- Address examples

- d **: deletes the all lines**

- 6d : deletes line 6

- /^$/d : deletes all blank lines

- 1,10d : deletes lines 1 through 10

- 1,/^$/d : deletes from line 1 through the first blank line

- /^$/,$d : deletes from the first blank line through the last line of the file

- /^$/,10d : deletes from the first blank line through line 10

- /^ya*y/,/[0-9]$/d : deletes from the first line that begins with yay, yaay, yaaay, etc. through the first line that ends with a digit

Plateforme Bioinfo Occitanie Toulouse

Genotoul Bioinfo

Hands-on `sed` deletion

- `sed "20,30d" fichier` : supprime les lignes 20 à 30 du fichier
- `sed "/toto/d" fichier` : supprime les lignes contenant la chaîne toto
- `sed "/toto/!d" fichier` : supprime toutes les lignes ne contenant pas la chaîne toto
- `sed "/^ *$/d" fichier` : supprime les lignes vides

  1,5!d would delete all lines except 1 through 5

Other useful functions

“print” - “= »

- p : By default, sed prints every line. When using "`sed -n`," it will not, by default, print any new lines. Using the "`p`" flag will cause the modified line to be printed. Without "`-n`", the line is duplicated
  - **1,5p**
    → will display lines 1 through 5
  - ***/^$/,$p***
    → will display the lines from the first blank line through the last line of the file
- = : prints the current line number to standard output.
  - Examples
    -
    -
    - `sed "/toto/=" fichier` : affiche le numéro de la ligne contenant la chaîne toto.

Other useful functions
`append – insert – change`

- Syntax for these functions is a little strange because they **must** be specified on multiple lines
- **append** *[address]a*
  text
- **insert** *[address]i*
  text
- **change** *[address(es)]c*
  text
- append/insert for single lines only, not range

Other useful functions

Differences : append - insert

- Append places *text* after the current line in pattern space
- Insert places *text* before the current line in pattern space
  - Each of these commands requires a **** following it. *text* must begin on the next line.
  - If text begins with whitespace, sed will discard it unless you start the line with a ****

```
>sed -e 'ajsdklfjslkjflskqjfdmlkj' < inputfile3.txt
one two three, one two three, one two three, one two three
jsdklfjslkjflskqjfdmlkj
four three two one, four three two one
jsdklfjslkjflskqjfdmlkj
one hundred, one hundred
jsdklfjslkjflskqjfdmlkj
one two three, one two three, one two three, one two three
jsdklfjslkjflskqjfdmlkj
four three two one, four three two one
jsdklfjslkjflskqjfdmlkj
one hundred, one hundred
jsdklfjslkjflskqjfdmlkj
>sed -e '1ajsdklfjslkjflskqjfdmlkj' < inputfile3.txt
one two three, one two three, one two three, one two three
jsdklfjslkjflskqjfdmlkj
four three two one, four three two one
one hundred, one hundred
one two three, one two three, one two three, one two three
four three two one, four three two one
one hundred, one hundred
```

```
>sed -e '1i              jsdklfjslkjflskqjfdmlkj' < inputfile3.txt
jsdklfjslkjflskqjfdmlkj
one two three, one two three, one two three, one two three
four three two one, four three two one
one hundred, one hundred
one two three, one two three, one two three, one two three
four three two one, four three two one
one hundred, one hundred
>sed -e 'i              jsdklfjslkjflskqjfdmlkj' < inputfile3.txt
jsdklfjslkjflskqjfdmlkj
one two three, one two three, one two three, one two three
jsdklfjslkjflskqjfdmlkj
four three two one, four three two one
jsdklfjslkjflskqjfdmlkj
one hundred, one hundred
jsdklfjslkjflskqjfdmlkj
one two three, one two three, one two three, one two three
jsdklfjslkjflskqjfdmlkj
four three two one, four three two one
jsdklfjslkjflskqjfdmlkj
one hundred, one hundred
```

Other useful functions

append – insert – change

- Unlike Insert and Append, Change can be applied to either a single line address or a range of addresses
- When applied to a range, the entire range is replaced by text specified with change, not each line
- What is the result of the following command ?

```
→ sed '1,2c aaacccc \
  > kkkk' fich2
```

```
cgaspin@belloc:~/TP/TP_REGEX$ cat fich2
$USA
$USSS
$US
$UAD
UAD-DUA
$UAD
```

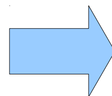
Other useful functions

append – insert – change

- Unlike Insert and Append, Change can be applied to either a single line address or a range of addresses
- When applied to a range, the entire range is replaced by text specified with change, not each line
- What is the result of the following command ?

→ `sed '1,2c aaacccc \`
`kkkk' fich2`

```
cgaspin@belloc:~/TP/TP_REGEX$ cat fich2
$USA
$USSS
$US
$UAD
UAD-DUA
$UAD
```



```
cgaspin@belloc:~/TP/TP_REGEX$ sed '1,2c aaacccc \
> kkkk' fich2
aaacccc
kkkk
$US
$UAD
UAD-DUA
$UAD
```

Other useful functions

Working with multiple lines

- The `N` command reads/appends the next line of input into the pattern space

- What are the results of the commands ?

→ `sed -e 'N' file`

→ `sed -e '{N; s/\n//}' file`

→ `sed -e '{N;N; s/\n//g}' file`

TP2

- Exercises

Plateforme Bioinfo Occitanie Toulouse

GenoToul Bioinfo

Part III
Programmable filter with `awk`

awk - General

Aho

Weinberger

Kernighan

- A general purpose programmable filter that handles text (strings) as easily as numbers
- Processes *fields* while **sed** only processes lines
- Gets input from
  - files
  - redirection and pipes
  - directly from standard input

Genotoul Bioinfo

awk - General

- A *pattern-action* language, like **sed**
- Looks a little like *C* but automatically handles input, field splitting, initialization, and memory management :
- A great prototyping language
  - `awk 'programme' file1 file2 …` : instructions between « `'` »
  - `awk 'programme'` : with standard input
  - `awk -f pfile file1 file2 ...`: instructions are in « `pfile` ».

awk - General

- A *pattern-action* language, like **sed**
- Looks a little like *C* but automatically handles input, field splitting, initialization, and memory management :
- A great prototyping language
  - `awk 'programme' file1 file2 …` : instructions between « `'` »
  - `awk 'programme'` : with standard input
  - `awk -f pfile file1 file2 ...`: instructions are in « `pfile` ».

Structure of an awk program

BEGIN { action } Optional BEGIN segment : for processing to run prior to reading input

pattern { action } pair(s) « pattern-action »: For each pattern matched, the corresponding {action} is done. At least one of the pattern or {action} has to be given. If pattern is not given, {action} is done for each line. Default *pattern* is to match all lines. If {action} is not given, the line is printed.

…

pattern { action }

END { action } Optionnel END segment: for processing to run at the end

- Search pattern in the input, line after line
- Run action on the line where pattern is found

```
awk '{print;}' FILENAME
```

→ Because pattern is not specified, each line will be printed

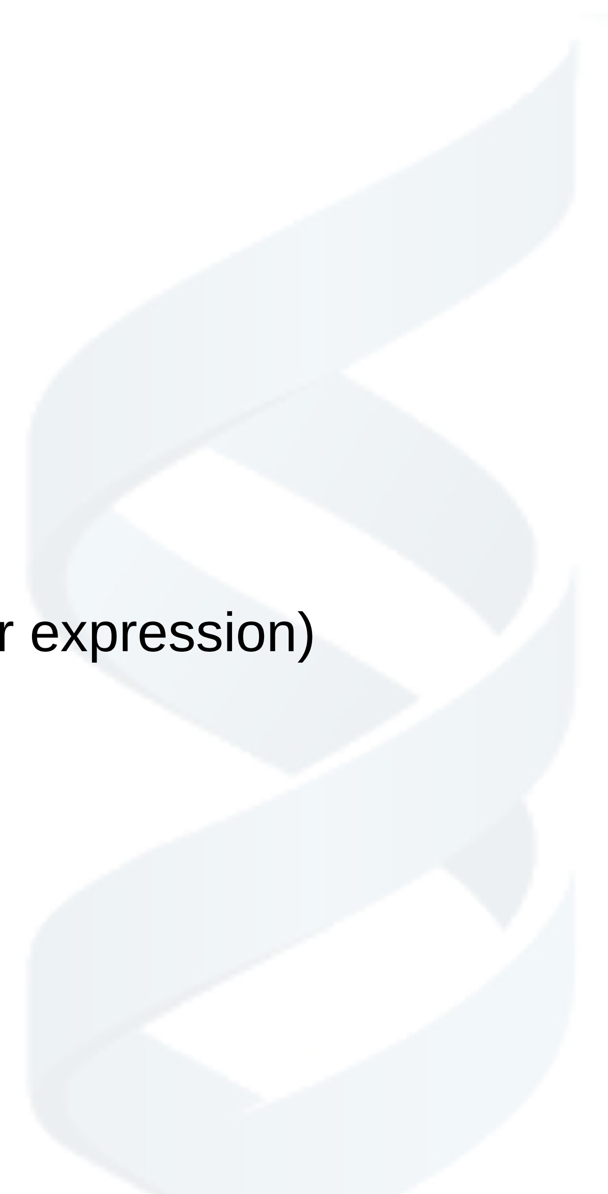
Patterns in awk

- « pattern » acts as a selector
- Patterns can be :
  - specific patterns : « BEGIN » or « END »
  - regular expression : /regexpr/
  - A predicate :   « x > 0 »
  - A combination of patterns with « && » or « || »
    - ***/NYU/*** matches if the string “NYU” is in the line
    - **x > 0** matches if the condition is true
    - ***/NYU/* && (x > 0)**

Operators in awk

Relational operators

- « **<** » : Less than
- « **<=** » : Less than or equal to
- « **>** » : Greater than
- « **>=** » : Greater than or equal to
- « **==** » : Equal to
- « **!=** » : Not equal to
- « **~** » : Matches (compares a string to a regular expression)
- « **!~** » : Does not match



Operators in awk

Logic operators

- « **&&** » : And (reports "true" if both sides are true)
- « **||** » : Or (reports "true" if either side, or both, are true)
- « **!** » : Not (Reverses true/false of the following expression)

Arithmetic operators

- « **+** » : Addition
- « **-** » : Subtraction
- « * » : Multiplication
- « **/** » : Division
- « **^** » : Exponentiation (** may also work)
- « **%** » : Remainder

Concatenation

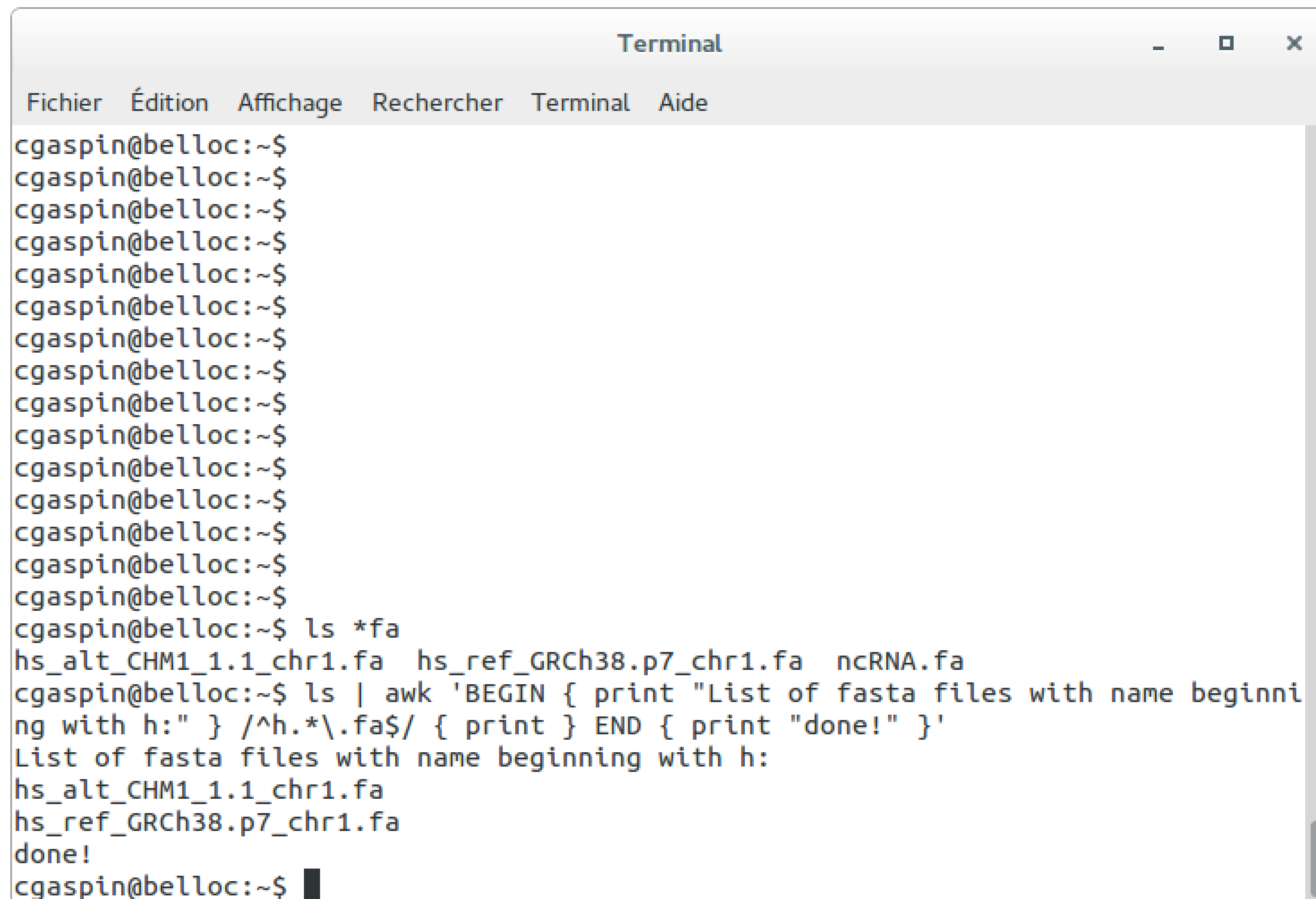
```
BEGIN {string = "Super" "power"; print string}
```

Actions in awk

- *action* may include a list of one or more C like statements, as well as arithmetic and string expressions and assignments and multiple output streams.
- *action* is performed on every line that matches *pattern*.
  - If *pattern* is not provided, *action* is performed on every input line
  - If *action* is not provided, all matching lines are sent to standard output.
- Since *patterns* and *actions* are optional, *actions* must be enclosed in braces to distinguish them from *pattern*.

Awk by example

```
ls | awk 'BEGIN { print "List of fasta files with name beginning with h:" } /^h.*\.fa$/ { print } END { print "done!" }'
```



awk - General

- Awk reconizes the concepts of « **file** », « **record** », and « **field** »
- A **file** consists of **records**, which by default are the lines of the file
- A **record** consists of **fields**, which by defaults are separated by spaces (or tabs)

awk - record

- Default record separator is **newline**
  - By default, **awk** processes its input a line at a time.
- Could be any other *regular expression*.
- **RS**: record separator
  - Can be changed in **BEGIN** action
- **NR** is the variable whose value is the number of the current record.

awk - field

- Each input line is split into fields.
  - **FS**: field separator: default is space (1 or more spaces or tabs)
  - **awk -F***c* option sets **FS** to the character *c*
    - Can also be changed in BEGIN
  - **$0** is the entire line
  - **$1** is the first field, **$2** is the second field, ….
- Only fields begin with **$**

awk - variable

- awk allows to define and use variables
  - BEGIN { **sum** = 0 } { **sum**++ } END { print **sum** }
- **awk** variables take on numeric (floating point) or string values according to context
- User-defined variables are *unadorned* (they need not be declared)
- By default, user-defined variables are initialized to the null string which has numerical value 0

awk - variable

- awk allows to define and use variables
  - BEGIN { **sum** = 0 } { **sum**++ } END { print **sum** }
- Some variables are predefined :
  - **RS**, **ORS**: record separators in input and output, respectively
  - **NR** : current number of records when processing
  - **FNR** : number of record in the file
  - **FS**, **OFS** : field separators in input and output, respectively
  - **NF** : Number of fields on the current record

awk by example

A program which counts the number of lines in `nf`:

```
awk 'BEGIN { total = 0 } { total ++ } END { print total }' nf
```

- `total` : a variable initialized to '0' at the beginning
- `total` : is incremented for each read
- The value of `total` is printed at the end

awk by example

Terminal

Fichier Édition Affichage Rechercher Terminal Aide

```
CTACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCCTAACCCCTAACCCTAACCCTAACCCTA
ACCCTAACCCTAACCCTAACCCCTAACCCTAACCCTAACCCTAACCCTCGCGGTACCCTCAGCCGGCCCG
CCCGCCCGGGTCTGACCTGAGGAGAACTGTGCTCCGCCTTCAGAGTACCACCGAAATCTGTGCAGAGGAC
AACGCAGCTCCGCCCTCGCGGTGCTCTCCGGGTCTGTGCTGAGGAGAACGCAACTCCGCCGTTGCAAAGG

==> ncRNA.fa <==
>HACA::KI0AEA23YO21AHM1:153-304
TCTAATAATCAAAGTTTGCTATTATATTGATGTAGTTACGAGACCGACCGTCTGTTTTATTGGTTAGATTTGAATTTAA
AAACATATTAGACTTCATGAAGGGTTTACTTAAAAATGCTCCTTTCAGGCTCCGATTTTACTTTTATTGGAA
>U6::KI0AEA4YJ21FM1:76-172
GCAGTTACTGCACTATTGATTTGGAACAACACTGAGAAGATTAGCAACGCCCCTGCGCAAGGACGGCGCTTTCTTTGAG
AGGTTGCCTGTTGTGAT
>U2::KI0ARB19YN08FM1:1720-1905
AAAAGTGACTAGTAAGATTGGAACAAACAACTTACTAGTCATGACTGATGTGATACAATCAAGAAAAAGCAAGCTTATT
CAACGATTGTGCATCAGCCATTTAACTCTGATGAGTTATTAACCAGATTTTACTCTGGCTAGAACAGATACTACACTTG
ATCAAAGCTATAAGCTTTGAGAGGTGA
>U6::KI0AFA4DA08FM1:355-451
GCAGTTACTGCACTATTGATTTGGAACAACACTGAGAAGATTAGCAACGCCCCTGCGCAAGGACGGCGCTTTCTTTGAG
AGGTTGCCTGTTGTGAT
>RNaseMRP_RNA::KI0AIA24YL10AHM1:633-793
GTAATCCCAAAGAACAGCTACGGAGTGCTTAGGCACTCTGGGGAAAGTCCTCGGACATACTGCAGAAATGCGTCCTGAA
AGGGGTATTCTAGTCATGTGATAAAGTTATGCTGTTAGGTATTGGTCTACTGAACACAATGAGGCTTACTCTTGGGAGT
AC
cgaspin@belloc:~$
```

```
awk -F":" '/>/ {print $4}' ncRNA.fa
```

- Field separator
- Pattern to search
- Action to perform if the pattern matches
- The file to operate upon

awk by example

- `awk 'NR==2,NR==5' FILE`

  → Write the lines from the second one to the fifth

- `awk 'NR==50' FILE`

  → write only the 50th line

- `awk 'NR < 26' FILE`

  → writhe the first 25 lines

- `awk 'NF > 0' FILE > NEWFILE`

  → Only non empty lines are printed, the result is in `NEWFILE`

- `awk 'NF > 4' FILE`

  → Only the lines with more than 4 fields are printed

- `awk 'END { print $NF }' FILE`

  → Only print the value of the last field in a record

Plateforme Bioinfo Occitanie Toulouse

Genotoul Bioinfo

awk by example

- What do the following commands ?
  - `awk 'NR > 25' FILE`
  - `awk 'END { print NR }' FILE`
  - `awk '{ print NF ":" $0 }’ FILE`
  - `awk '{ print NR ":" $0 }' FILE`
  - `awk '$5 == "abc123"' FILE`
  - `awk '{ print $1, $2 }' FILE`
  - `awk '{ print $2, $1 }' FILE`
  - `awk '{ $2 = ""; print }' FILE`
  - `awk '/REGEX/' FILE`
  - `awk '!/REGEX/' FILE`
  - `awk '/AAA|BBB|CCC/' FILE`
  - `awk ’{print NF, $1 }’ FILE`

Functions on strings

- `length(s)`: give the length of the string. Default `s` is $0.

```
awk 'BEGIN{nc=0}{nc=nc+length; nw=nw+NF} END {print
NR,"lines,",nw,"words,",nc,"chars"}' FILE
```

- `substr(s,i,len)`: returns the substring of string `s`, starting at index `i` of length `len`. If length is omitted, the suffix of `s` starting at index `i` is returned.
- `sub(s,sub,string)`: performs single substitution. It replaces the first occurrence of `s` with `sub`. The third parameter is optional. If it is omitted, $0 is used.
- `tolower(s)`: returns a copy of string `s` with all upper-case characters converted to lower-case.
- `toupper(s)`: returns a copy of string `s` with all upper-case characters converted to upper-case.
- `split(s,arr,regex)`: This function splits the string `s` into fields by regular expression `regex` and the fields are loaded into the array `arr`. If regex is omitted, then `FS` is used.

Genotoul Bioinfo

Loops statements

- If

```
awk 'BEGIN {num = 10; if (num % 2 == 0) printf "%d is even
number.\n", num }'
```

- If-Then-Else

```
awk 'BEGIN {num = 11; if (num % 2 == 0) printf "%d is even
number.\n", num; else printf "%d is odd number.\n", num }'
```

Control flow statements

- **for** loop

```
awk 'BEGIN {for (i = 1;i <= 5;++i) print I}'
```

```
echo "1 2 3 4"|awk '{for (i=1;i<=NF;i++) total=total+$i}; END
{print total}'
```

- **while** loop

```
awk 'BEGIN {i= 1;while (i < 6) {print i;++i}}'
```

```
awk 'BEGIN{while(1) print "forever"}'
```

```
awk 'BEGIN{x=1;while(1){print "Iteration";if (x==10) break;x++;}}'
```

- **Do-while loop**

```
awk 'BEGIN {i = 1; do {print i; ++i} while (i < 6)}'
```

```
awk 'BEGIN{count=1;do print "This gets printed at least once";
while(count!=1)}'
```

Control flow statements

- If

```
awk 'BEGIN {num = 10; if (num % 2 == 0) printf "%d is even
number.\n", num }'
```

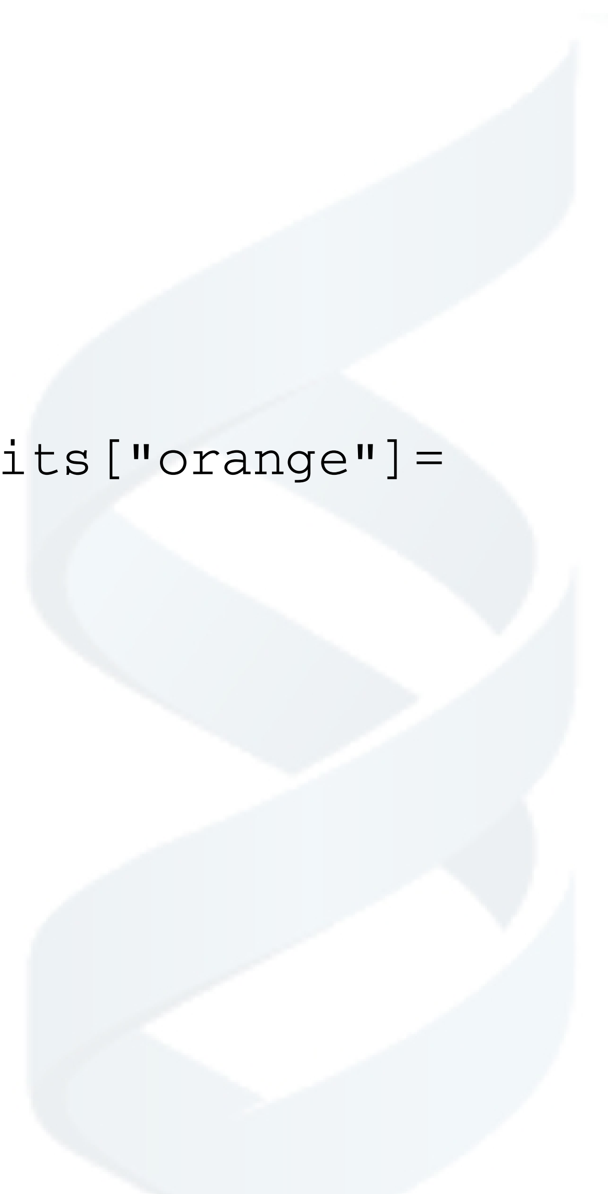
- If-Then-Else

```
awk 'BEGIN {num = 11; if (num % 2 == 0) printf "%d is even
number.\n", num; else printf "%d is odd number.\n", num }'
```

Arrays in awk

- Array elements are not declared
- Array can have ***any*** value:
  - Numbers
  - Strings (***associative arrays***)
- Example

```
awk 'BEGIN {fruits["mango"]="yellow";fruits["orange"]=
"orange";print fruits["orange"]}'
```



awk by example

- What do the following programs ?

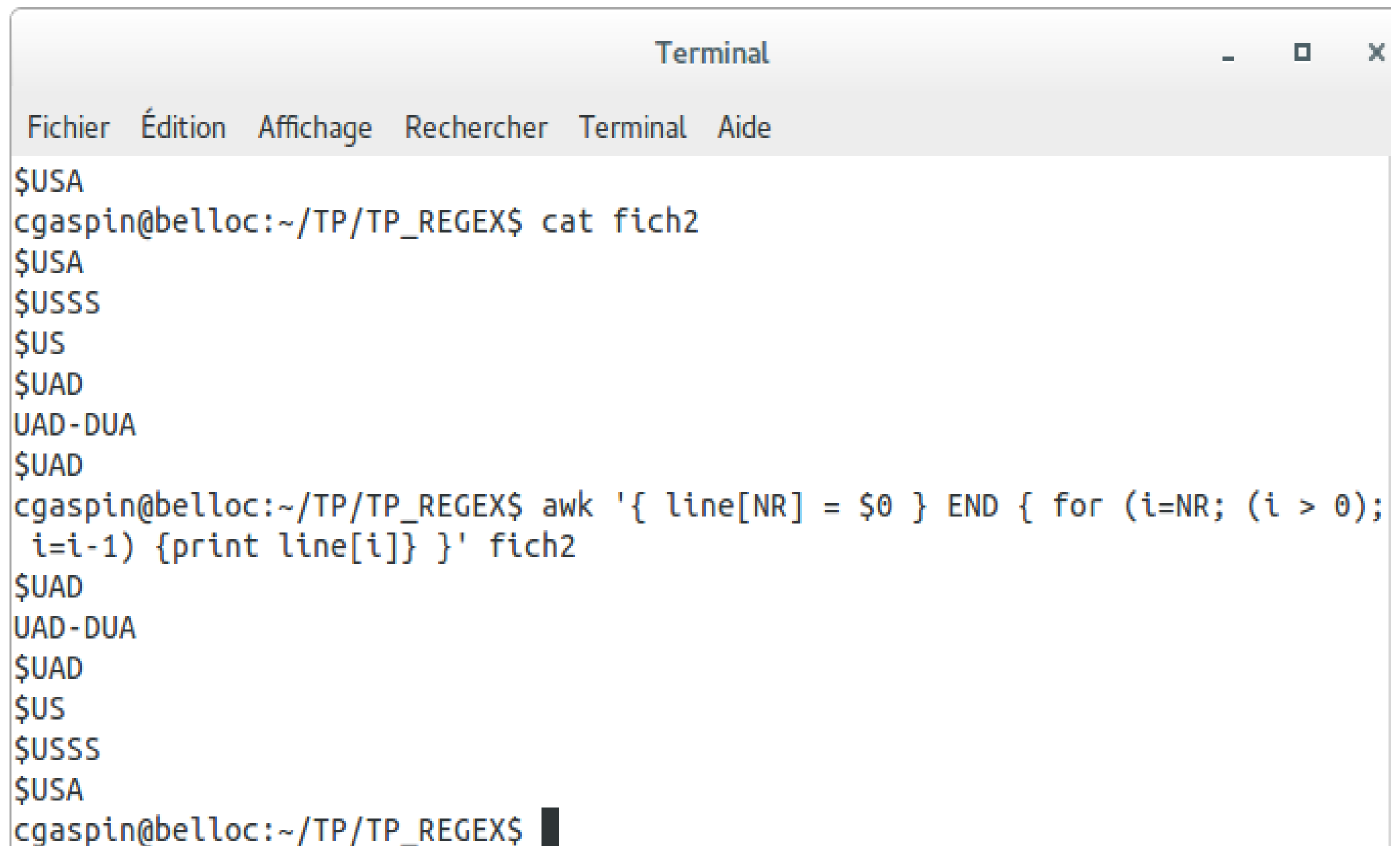
```
awk '{line[NR]=$0} END {for (i=NR;(i>0);i=i-1) {print
line[i]}}' file
```

```
Awk 'BEGIN {item[101]="HD
Camcorder";item[102]="Refrigerator";item[103]="MP3
Player";item[104]="Tennis Racket";item[105]="Laser
Printer";item[1001]="Tennis Ball";item[55]="Laptop";
item["na"]="Not Available"; for (x in item) print item[x];}
```

awk by example

- What do the following program ?

```
awk '{line[NR]=$0} END {for (i=NR;(i>0);i=i-1) {print
line[i]}}' file
```



awk by example

- What can you do with awk ?

Question files : give the list of chromosomes in file `file1.bed`

chr2	74711	127472363	Pos1	0	+
chr3	74723	127473530	Pos2	0	+
chr1	73530	127474697	Pos3	0	+
chr2	17469	127475864	Pos4	0	+
chr3	12747	127477031	Neg1	0	-
chr5	17477	127478198	Neg2	0	-
chr7	74781	127479365	Neg3	0	-
chr7	74795	127480532	Pos5	0	+
chr1	12748	127481699	Neg4	0	-

```
awk '{print $1}' file1.bed | sort | uniq
```

Part III

TP3

- Exercises

Genotoul Bioinfo

Plateforme Bioinfo Occitanie Toulouse

Used material

- https://www.powershow.com/view1/18c06e-ZDc1Z/CSCI_20330_20The_20UNIX_20System_powerpoint_ppt_presentation
- http://www.grymoire.com/Unix/Sed.html#uh-5
- https://www.tutorialspoint.com/awk/awk_quick_guide.htm