Initiation à l'environnement Unix
CM sur `grep`, `sed`, `awk` et les expressions rationnelles

Pierre Rousselin

Université Sorbonne Paris Nord
L1 informatique et DL
décembre 2024

Introduction à `grep` et aux expressions rationnelles

Introduction à `sed`

Introduction à `awk`

Expressions rationelles étendues

Introduction à `grep` et aux expressions rationnelles

Introduction à `sed`

Introduction à `awk`

Expressions rationelles étendues

La commande `grep`

`g/re/p`

Afficher les lignes des fichiers en argument (ou de l'entrée standard si pas d'argument fichier) correspondant à *une expression rationnelle*.

- `g` pour *global* : on cherche dans tout le fichier ;
- `re` pour *regular expression* : lignes correspondant à un certain motif ;
- `p` pour *print* : afficher sur la sortie standard.

La commande `grep`

`g/re/p`

Afficher les lignes des fichiers en argument (ou de l'entrée standard si pas d'argument fichier) correspondant à *une expression rationnelle*.

- `g` pour *global* : on cherche dans tout le fichier ;
- `re` pour *regular expression* : lignes correspondant à un certain motif ;
- `p` pour *print* : afficher sur la sortie standard.

Conseil : si votre version de `grep` a une option qui colore les correspondances, il vaut mieux l'utiliser pendant la phase d'apprentissage.
Avec `GNU grep` : `--color=auto`
On peut utiliser un alias (qu'on peut mettre dans son `~/.bashrc`).

```
$ alias grep='grep --color=auto'
```

grep : exemples

```
$ cat poule
Une poule sur un mur
Qui picore du pain dur
Picoti, Picota
Lève la queue et puis s'en va
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ grep , poule poissons
poule:Picoti, Picota
poissons:Nagent, nagent
poissons:Nagent, nagent, nagent
$ grep 'poisson' poule poissons
poissons:Les petits poissons dans l'eau
poissons:Les petits poissons dans l'eau
$ grep -c 'nagent' poule poissons
poule:0
poissons:2
$ grep -v 'e' poule poissons
poule:Picoti, Picota
$
```

grep : exemples

```
$ cat poule
Une poule sur un mur
Qui picore du pain dur
Picoti, Picota
Lève la queue et puis s'en va
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ grep , poule poissons
poule:Picoti, Picota
poissons:Nagent, nagent
poissons:Nagent, nagent, nagent
$ grep 'poisson' poule poissons
poissons:Les petits poissons dans l'eau
poissons:Les petits poissons dans l'eau
$ grep -c 'nagent' poule poissons
poule:0
poissons:2
$ grep -v 'e' poule poissons
poule:Picoti, Picota
$
```

Option `-c` pour *compter*, `-v` pour *inverser* la recherche.

Si plusieurs fichiers en argument, le nom du fichier suivi du caractère `:` préfixe l'affichage des lignes.

grep : exemples

```
$ cat poule
Une poule sur un mur
Qui picore du pain dur
Picoti, Picota
Lève la queue et puis s'en va
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ grep '[v-zV-Z]' poule poissons
poule:Lève la queue et puis s'en va
$ grep ' [aeiou]' poule poissons
poule:Une poule sur un mur
poule:Lève la queue et puis s'en va
poissons:Nagent aussi bien que les gros
$ grep ' [Pp]...t' poule poissons
poule:Picoti, Picota
poissons:Les petits poissons dans l'eau
poissons:Les petits poissons dans l'eau
$ grep '^[AEIOU]' poule poissons
poule:Une poule sur un mur
$ grep '^[^N-R]' poule poissons
poule:Une poule sur un mur
poule:Lève la queue et puis s'en va
poissons:Les petits poissons dans l'eau
poissons:Les petits poissons dans l'eau
$
```

grep : exemples

```
$ cat poule
Une poule sur un mur
Qui picore du pain dur
Picoti, Picota
Lève la queue et puis s'en va
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ grep ' [^ ][^ ][^ ]$' poule poissons
poule:Une poule sur un mur
poule:Qui picore du pain dur
$ grep ',.......,' poule poissons
poissons:Nagent, nagent, nagent
$ grep ',.*,' poule poissons
poissons:Nagent, nagent, nagent
$ grep '[aeiou][^ ]*[aeiou][^ ]*[aeiou]' poule poissons
poule:Une poule sur un mur
poule:Qui picore du pain dur
poule:Picoti, Picota
poule:Lève la queue et puis s'en va
poissons:Les petits poissons dans l'eau
poissons:Les petits poissons dans l'eau
poissons:Nagent aussi bien que les gros
$ grep 'e.*e.*e.*e' poule poissons
poule:Lève la queue et puis s'en va
poissons:Nagent aussi bien que les gros
$
```

Expressions rationnelles

- Le premier argument (non option) de grep est une *expression rationnelle* (le nom vient des automates et de la théorie des langages, voir le cours de L3).
- On dit aussi *expression régulière* et en anglais *regular expression*, abrégé en *regexp* ou *regex*.
- Une expression rationnelle décrit les chaînes de caractères qui *lui correspondent*, en anglais (et de plus en plus en français aussi) qui *match*.
- Ne pas confondre *motif shell* et *expression rationnelle*, même si ça se ressemble beaucoup !

Expressions rationnelles

- Le premier argument (non option) de grep est une *expression rationnelle* (le nom vient des automates et de la théorie des langages, voir le cours de L3).
- On dit aussi *expression régulière* et en anglais *regular expression*, abrégé en *regexp* ou *regex*.
- Une expression rationnelle décrit les chaînes de caractères qui *lui correspondent*, en anglais (et de plus en plus en français aussi) qui *match*.
- Ne pas confondre *motif shell* et *expression rationnelle*, même si ça se ressemble beaucoup !
- motif shell $\neq$ expression rationnelle

Expressions rationnelles

- Le premier argument (non option) de grep est une *expression rationnelle* (le nom vient des automates et de la théorie des langages, voir le cours de L3).
- On dit aussi *expression régulière* et en anglais *regular expression*, abrégé en *regexp* ou *regex*.
- Une expression rationnelle décrit les chaînes de caractères qui *lui correspondent*, en anglais (et de plus en plus en français aussi) qui *match*.
- Ne pas confondre *motif shell* et *expression rationnelle*, même si ça se ressemble beaucoup !
- motif shell $\neq$ expression rationnelle
- motif shell $\neq$ expression rationnelle

Métacaractères

Dans une expression régulière (sans entrer trop dans le détail)

- Les caractères `.` `*` `[` `^` `$` `\` sont spéciaux (on dit que ce sont des *métacaractères*).
- Les autres caractères se représentent eux-mêmes, on dit qu'ils sont *littéraux*.
- Le point `.` est un *joker*, il représente **un** caractère quelconque.
- Un ensemble de caractères entre crochets représente **un et un seul de ces caractères**.
  - On peut utiliser des suite, par exemple `[0-9]` ou `[A-E]`.
  - Un accent circonflexe en début d'ensemble sert de négation, par exemple `[^0-9]` est **un** caractère qui n'est pas un chiffre.
  - Les règles sont les mêmes que pour les crochets dans les motifs shell en remplaçant le point d'exclamation `!` par l'accent circonflexe `^` pour la négation. Voir début du CM2.

Métacaractères

Dans une expression rationnelle (sans entrer trop dans les détails)

- L'étoile (`*`) signifie *pour le caractère ou métacaractère précédent* : 0, 1, 2, ... fois.
  Par exemple, l'expression rationnelle `ab*c` correspond à `ac`, `abc`, `abbc`, ...
- La contre-oblique (\) rend littéral un métacaractère qui la suit.
  Par exemple, l'expression rationnelle `.*\.c` correspond à une chaîne qui se termine par un point puis un `c`.
- L'accent circonflexe (`^`) et le caractère dollar (`$`) sont spéciaux respectivement en début et en fin d'expression. On les appelle des *ancres* (en anglais *anchors*). Ils correspondent respectivement au début et à la fin de la ligne.

Expressions rationnelles restreintes

`.` `[` `*` `^` `$` `\`

Appelons (dans ce cours) expressions rationnelles *restreintes* les expressions formées avec :

- les caractères littéraux et les métacaractères inhibés par une contre-oblique ;
- le caractère joker `.` ;
- les expressions crochet `[...]` et `[^...]` ;
- l'opérateur `*` ;
- les ancres `^` et `$`.

Différences et similarités entre motifs shell et expressions rationnelles

Motif shell	regexp	correspondances
`?`	`.`	un car. quelconque
`[...]`	`[...]`	un des car. dans les crochets
`[!...]`	`[^...]`	un car. qui n'est pas dans les crochets
`*`	`.*`	une chaîne quelconque
Pas d'equivalent	`*`	répétition du car. précédent
Pas d'equivalent	`^` et `$`	début/fin de ligne
Pas d'equivalent	Autres car. spéciaux	(voir compléments)

Exemples d'expressions rationnelles restreintes

- `[Ss]hell` correspond à `shell` ou à `Shell`.
- `[0-9][0-9][0-9][0-9][0-9]` correspond à

Exemples d'expressions rationnelles restreintes

- `[Ss]hell` correspond à `shell` ou à `Shell`.
- `[0-9][0-9][0-9][0-9][0-9]` correspond à un nombre décimal à 5 chiffres (par exemple un code postal).
- `[a-zA-Z_][0-9a-zA-Z_]*` correspond à

Exemples d'expressions rationnelles restreintes

- `[Ss]hell` correspond à `shell` ou à `Shell`.
- `[0-9][0-9][0-9][0-9][0-9]` correspond à un nombre décimal à 5 chiffres (par exemple un code postal).
- `[a-zA-Z_][0-9a-zA-Z_]*` correspond à une chaîne non vide ne contenant que des lettres minuscules ou majuscules, des chiffres et des soulignés (*underscore*), ne commençant pas par un chiffre (bref un nom de variable correct en shell ou en C).
- `^$` correspond à

Exemples d'expressions rationnelles restreintes

- `[Ss]hell` correspond à `shell` ou à `Shell`.
- `[0-9][0-9][0-9][0-9][0-9]` correspond à un nombre décimal à 5 chiffres (par exemple un code postal).
- `[a-zA-Z_][0-9a-zA-Z_]*` correspond à une chaîne non vide ne contenant que des lettres minuscules ou majuscules, des chiffres et des soulignés (*underscore*), ne commençant pas par un chiffre (bref un nom de variable correct en shell ou en C).
- `^$` correspond à une ligne vide.
- `^..*$` correspond à

Exemples d'expressions rationnelles restreintes

- `[Ss]hell` correspond à `shell` ou à `Shell`.
- `[0-9][0-9][0-9][0-9][0-9]` correspond à un nombre décimal à 5 chiffres (par exemple un code postal).
- `[a-zA-Z_][0-9a-zA-Z_]*` correspond à une chaîne non vide ne contenant que des lettres minuscules ou majuscules, des chiffres et des soulignés (*underscore*), ne commençant pas par un chiffre (bref un nom de variable correct en shell ou en C).
- `^$` correspond à une ligne vide.
- `^..*$` correspond à une ligne non vide.
- `^[aeiouy]*$` correspond à

Exemples d'expressions rationnelles restreintes

- `[Ss]hell` correspond à `shell` ou à `Shell`.
- `[0-9][0-9][0-9][0-9][0-9]` correspond à un nombre décimal à 5 chiffres (par exemple un code postal).
- `[a-zA-Z_][0-9a-zA-Z_]*` correspond à une chaîne non vide ne contenant que des lettres minuscules ou majuscules, des chiffres et des soulignés (*underscore*), ne commençant pas par un chiffre (bref un nom de variable correct en shell ou en C).
- `^$` correspond à une ligne vide.
- `^..*$` correspond à une ligne non vide.
- `^[aeiouy]*$` correspond à une ligne ne contenant que des voyelles (éventuellement vide).

grep : version très allégée

`grep [-c] [-inv] motif [fichier]...`

Affiche les lignes du ou des fichiers donnés en argument (de l'entrée standard si aucun fichier n'est donné comme argument) **dont une sous-chaîne** correspond à l'expression rationnelle `motif`.
Sauf dans les cas les plus simples, `motif` risque de contenir des caractères qui sont spéciaux *pour le shell*, penser à les inhiber !

- `-c` : **C**ompte les lignes qui correspondent au lieu de les afficher
- `-v` : in**V**erse la recherche (affiche les lignes ne correspondant pas au motif)
- `-i` : **I**gnore la casse (c'est-à-dire ne pas faire de différence entre majuscules et minuscules)
- `-n` : précéder chaque ligne qui correspond au motif de son **N**uméro de ligne
- `-q` : **Q**uiet, pour n'utiliser que le statut de sortie (`if echo "$ligne" | grep -q lol; then ... fi`).

Introduction à `grep` et aux expressions rationnelles

Introduction à `sed`

Introduction à `awk`

Expressions rationelles étendues

`sed`

- `sed` signifie *Stream EDitor*, c'est-à-dire « éditeur de flux » ;
- La commande `sed` lit les lignes des fichiers donnés en argument (de l'entrée standard s'il n'y en a pas) **unes par unes** : imaginer un ruisseau qui coule, emportant avec lui des lignes de texte.
- Chaque ligne, suivant qu'elle vérifie un certain critère appelé *adresse* va déclencher un certain nombre *d'actions*.
- Une fois toutes les actions effectuées (il peut n'y en avoir aucune), la ligne va être imprimée *sur la sortie standard* (sauf si elle est supprimée et/ou que l'option `-n` est donnée à `sed`).

sed : exemples

Exemple avec

- action `d` pour *delete* et
- zéro, une ou deux adresses numériques.

```
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ sed 3d poissons
Les petits poissons dans l'eau
Nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
$ sed 2,4d poissons
Les petits poissons dans l'eau
Nagent aussi bien que les gros
$ sed '2,4 d' poissons
Les petits poissons dans l'eau
Nagent aussi bien que les gros
$ sed '3,$ d' poissons
Les petits poissons dans l'eau
Nagent, nagent
$ sed d poissons # 0 adresse
$
```

sed : exemples

Exemple avec

- action `d` pour *delete* et
- une ou deux adresses numériques et/ou expressions rationnelles.

```
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ sed '/petits/ d' poissons
Nagent, nagent
Nagent, nagent, nagent
Nagent aussi bien que les gros
$ sed '/,.*,/ d' poissons
Les petits poissons dans l'eau
Nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
$ sed '/,.*,/,/poissons/ d' poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent aussi bien que les gros
$ sed '1,/[nN]agent/ d' poissons
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

Adresses

- 0 adresse : l'action concerne toutes les lignes.
- 1 adresse numérique `m` : l'action ne concerne que la ligne numéro m.
- 1 adresse expression rationnelle `/regexp/` : l'action ne concerne que les lignes correspondant à l'expression rationnelle `regexp`.
- 2 adresses numériques `m,n` : l'action ne concerne que les lignes de numéro compris entre m et n.
- Le dollar (`$`) désigne la dernière ligne.
- 2 adresses expressions rationnelles `/regexp1/,/regexp2/` : à partir de la première ligne correspondant à `regexp1` jusqu'à la prochaine ligne correspondant à `regexp2`, puis de nouveau de la prochaine ligne correspondant à `regexp1` jusqu'à ...
- 2 adresses avec un numéro de ligne et une expression rationnelle, ...

sed : exemples

- L'action `q` (*quit*) entraîne la fin du processus `sed`.
- L'option `-n` désactive l'impression automatique des lignes.
- L'action `p` (*print*) affiche la ligne courante.

```
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ sed 2q poissons
Les petits poissons dans l'eau
Nagent, nagent
$ sed '/,.*,/ q' poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
$ sed '3,4 p' poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Les petits poissons dans l'eau
Nagent aussi bien que les gros
$ sed -n '/,.*,/,/poissons/ p' poissons
Nagent, nagent, nagent
Les petits poissons dans l'eau
```

sed : l'action s

Action la plus courante : `s` pour *substitute* (« remplacer »).

```
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ sed 's/[Nn]agent/volent/' poissons
Les petits poissons dans l'eau
volent, nagent
volent, nagent, nagent
Les petits poissons dans l'eau
volent aussi bien que les gros
$ sed '$ s/[Nn]agent/volent/' poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
volent aussi bien que les gros
$ sed '2,3 s/[Nn]agent/volent/' poissons
Les petits poissons dans l'eau
volent, nagent
volent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

sed : l'action s

Les drapeaux (*flags*) `g` (*global*), `p` (*print*) et 1, 2, ... (numéro de la correspondance). [1]

```
$ cat poissons
Les petits poissons dans l'eau
Nagent, nagent
Nagent, nagent, nagent
Les petits poissons dans l'eau
Nagent aussi bien que les gros
```

```
$ sed 's/[Nn]agent/volent/g' poissons
Les petits poissons dans l'eau
volent, volent
volent, volent, volent
Les petits poissons dans l'eau
volent aussi bien que les gros
$ sed -n 's/poissons/oiseaux/p' poissons
Les petits oiseaux dans l'eau
Les petits oiseaux dans l'eau
$ sed -n 's/[Nn]agent/volent/pg' poissons
volent, volent
volent, volent, volent
volent aussi bien que les gros
$ sed -n 's/[Nn]agent/volent/2p' poissons
Nagent, volent
Nagent, volent, nagent
```

1. Le drapeau 1 est le cas par défaut (première correspondance).

Quelle partie de la ligne « matche » ?

```
$ sed -n 's/[Nn].*t/volent/gp' poissons
volent
volent
volent aussi bien que les gros
$ sed -n 's/[^ ]*/MOT/gp' poissons
MOT MOT MOT MOT MOT
MOT MOT
MOT MOT MOT
MOT MOT MOT MOT MOT
MOT MOT MOT MOT MOT MOT
$ sed -n 's/.*,/blah/gp' poissons
blah nagent
blah nagent
$ sed -n 's/[^ ,]*,/blah/gp' poissons
blah nagent
blah blah nagent
```

Quelle partie de la ligne « matche » ?

La chaîne qui correspond à l'expression rationnelle est, par ordre de priorité :

- la plus à gauche possible puis
- la plus longue possible.

Puis, si besoin (drapeau g ou drapeau nombre > 1), on regarde, à partir de la fin de cette première chaîne les correspondances suivantes.

`sed` version allégée 0%

```
sed [-n] prog_sed [fichier]...
```

- L'argument `prog_sed` est formé :
  - d'une partie *adresse* (éventuellement vide) ;
  - d'une partie *action* (nous avons vu `d`, `q`, `p` et `s`).
- `sed` lit une par une chaque ligne des fichiers (ou de l'entrée standard si il n'y a pas de fichier) ;
- si la ligne correspond à la partie *adresse* de `prog_sed`, la partie *action* de `prog_sed` a lieu ;
- puis la ligne (éventuellement transformée) est écrite sur la sortie standard (sauf si option `-n` ou action `d`).

Action la plus courante : `s` pour *substitute*.

```
s/regexp/remplacement/drapeaux
```

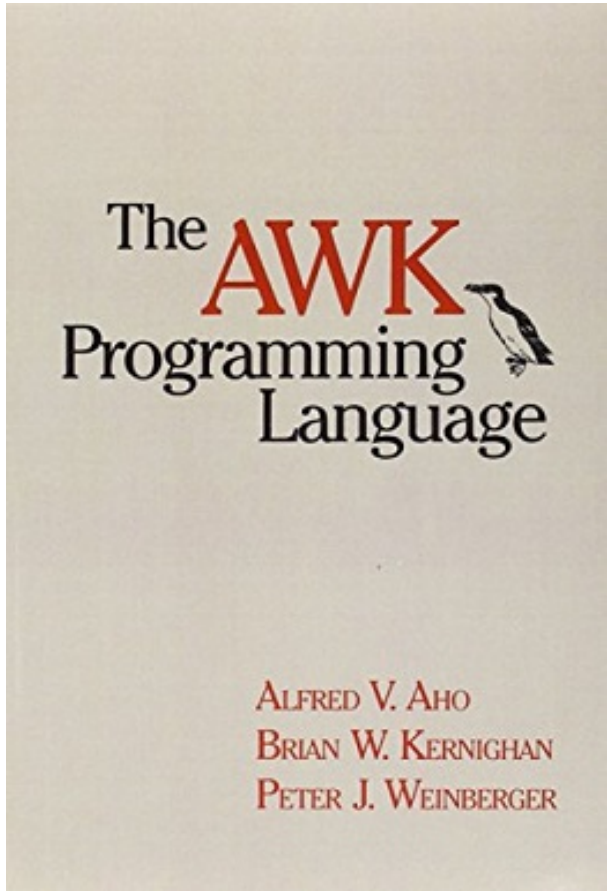
Introduction à `grep` et aux expressions rationnelles

Introduction à `sed`

Introduction à `awk`

Expressions rationelles étendues

Aho, Weinberger, Kernighan



Livre publié en 1988 chez Addison-Wesley, écrit par les inventeurs du langage.
Comme tous les livres de Kernighan, il est exceptionnel.

La commande `awk` (version allégée)

```
awk [-F séparateurs] prgrm_awk [fichier]...
```

Le programme en langage `awk` a la forme suivante :

```
BEGIN { action }
critère1 { action }
...
critèreN { action }
END { action }
```

Les actions suivant le critère `BEGIN` ont lieu au début du processus et celles suivant `END` à la fin. En-dehors de ça, le principe est le même que pour `sed` :

- chaque ligne du ou des fichiers données en argument (de l'entrée standard s'il n'y en a pas) est lue, et **séparée en champs** par awk, puis
- pour chacun des `critères`, si elle le remplit, les actions associées s'opèrent.

Quelques *one-liners* en `awk`

```
$ cat poule
Une poule sur un mur
Qui picore du pain dur
Picoti, Picota
Lève la queue et puis s'en va
```

```
$ awk '{ print $1 }' poule
Une
Qui
Picoti,
Lève
$ awk '{ print NF }' poule
5
5
2
7
$ awk '{ print NR, $0 }' poule
1 Une poule sur un mur
2 Qui picore du pain dur
3 Picoti, Picota
4 Lève la queue et puis s'en va
$ awk '/^P/ { print }' poule
Picoti, Picota
$ awk '/^P/' poule # action par défaut
Picoti, Picota
```

Rudiments de `awk` (1)

- Par défaut, les champs sont séparés par des blancs (tabulations et/ou espaces).
- Quelques variables prédéfinies :
  - `$1`, `$2`, ... : les valeurs des différents champs de la ligne
  - `$0` : la ligne entière
  - `NF` : *number of fields*, nombre de champs de la ligne
  - `NR` : *number of records*, numéro de la ligne

Rudiments de `awk` (1)

- Par défaut, les champs sont séparés par des blancs (tabulations et/ou espaces).
- Quelques variables prédéfinies :
  - `$1`, `$2`, ... : les valeurs des différents champs de la ligne
  - `$0` : la ligne entière
  - `NF` : *number of fields*, nombre de champs de la ligne
  - `NR` : *number of records*, numéro de la ligne
- Exemple de condition : `/regexp/`, vraie si la ligne (c'est-à-dire une sous-chaîne de la ligne) correspond à l'expression rationnelle `regexp`.
- Condition vide : l'action s'applique à toutes les lignes.
- Action vide : action `print` par défaut (imprime la ligne entière, comme `print $0`).

Quelques *one-liners* en awk

```
$ awk '{ print $NF, $1 }' poule
mur Une
dur Qui
Picota Picoti,
va Lève
$ awk 'NR % 2 == 0' poule
Qui picore du pain dur
Lève la queue et puis s'en va
$ awk 'NR % 2 == 0 && /^Q/' poule
Qui picore du pain dur
$ awk '{ $1 = toupper($1); print $0 }' poule
UNE poule sur un mur
QUI picore du pain dur
PICOTI, Picota
LÈVE la queue et puis s'en va
$ awk '{ $(NF+1) = "lalala"; print $0 }' poule
Une poule sur un mur lalala
Qui picore du pain dur lalala
Picoti, Picota lalala
Lève la queue et puis s'en va lalala
```

Rudiments de `awk` (2)

- L'opérateur `$` d'accès au champ peut être suivi d'une expression variable (par exemple `NF` ou `NF+1`).
- Les champs, et même le nombre de champs sont modifiables.
- Quelques fonctions prédéfinies de `awk` :
  - `tolower(s)`, `toupper(s)`, retourne la chaîne `s` mise en caractères minuscules (respectivement majuscules).
  - `length(s)` : retourne la longueur de la chaîne `s`.
  - `rand()` : retourne un nombre pseudo-aléatoire entre 0 et 1.
  - ... une vingtaine d'autres fonctions très pratiques.
- La condition peut être une expression arithmétique faisant apparaître les variables prédéfinies (par exemple `NF > 3` pour sélectionner les lignes ayant au moins 4 champs). La syntaxe est très proche de celle du C.

Avec BEGIN et END

```
$ awk 'BEGIN { print "LA COMPTINE DE LA POULE" }
> { print }
> END { print "FIN DE CETTE COMPTINE" }' poule
LA COMPTINE DE LA POULE
Une poule sur un mur
Qui picore du pain dur
Picoti, Picota
Lève la queue et puis s'en va
FIN DE CETTE COMPTINE
$ awk '{ somme += NF }
> END { print somme }' poule
19
```

Rudiments de `awk` (3)

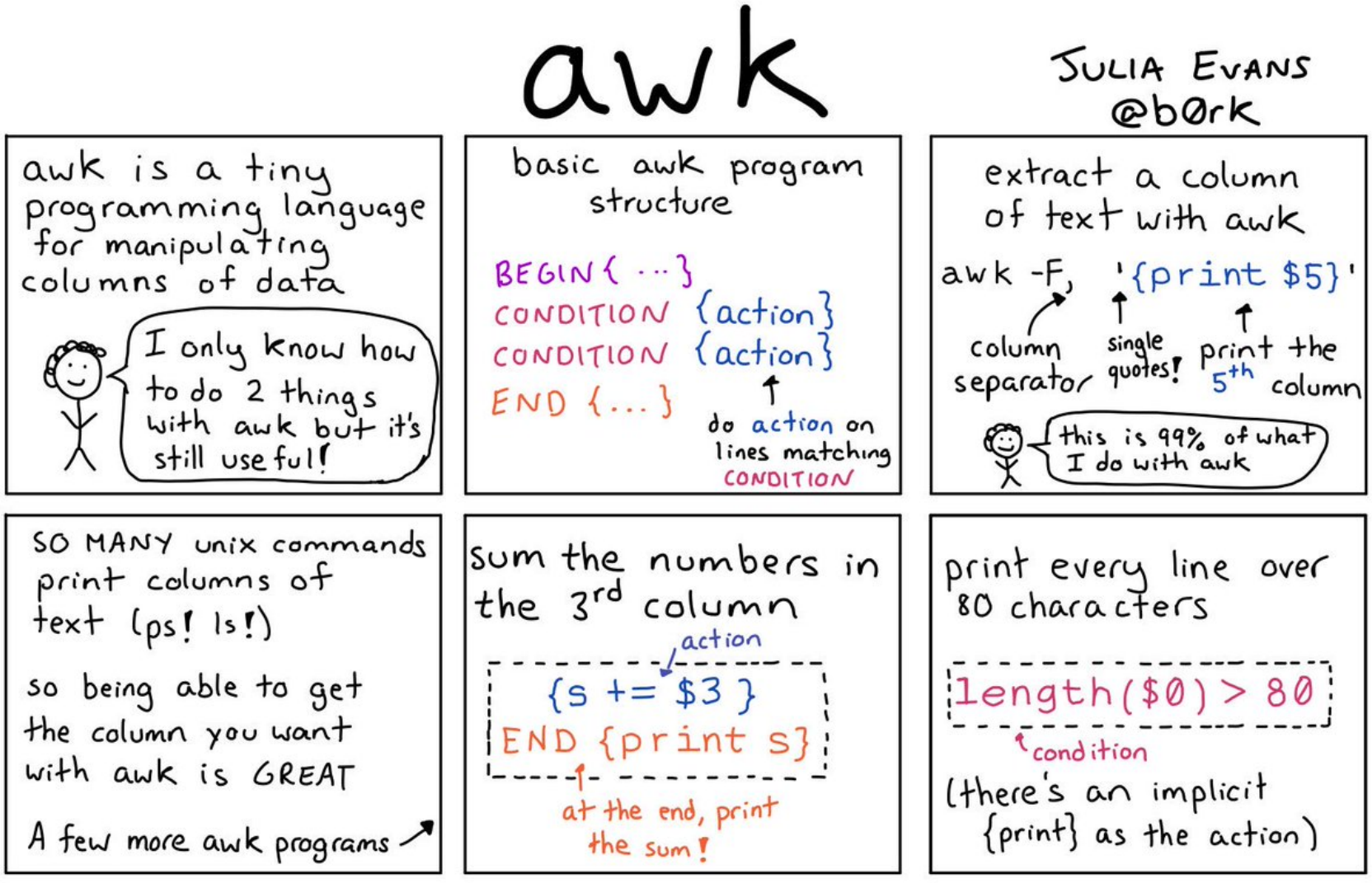
- Les variables n'ont pas besoin d'être déclarées.
- Les variables sont interprétées comme des chaînes ou des nombres (à virgule flottante) suivant le contexte : `awk` « devine » quel type utiliser.
- Les variables numériques sont initialisées à 0 et les variables chaînes à la chaîne vide.

Autre séparateur

```
$ cat informaticiens.csv
Bellman,Richard,1920,1984,Programmation dynamique
Church,Alonzo,1903,1995,lambda-calcul
Codd,Edgar Frank,1923,2003,Bases de données relationnelles
Dijkstra,Edsger Wybe,1930,2002,Algorithme de Dijkstra:sémaphores
$ awk -F, '{ print $2, $1 }' informaticiens.csv
Richard Bellman
Alonzo Church
Edgar Frank Codd
Edsger Wybe Dijkstra
$ awk -F, '{ printf("%s %s : %d\n", $2, $1, $4 - $3) }' informaticiens.
Richard Bellman : 64
Alonzo Church : 92
Edgar Frank Codd : 80
Edsger Wybe Dijkstra : 72
```

Le langage `awk`

- `awk` est un petit langage de programmation ;
- élégant et très simple d'utilisation ;
- idéal pour travailler sur des fichiers de données séparés en champs ;
- on peut l'utiliser :
  - directement en ligne de commande ;
  - dans des scripts shell ;
  - ou même écrire des programmes complets en `awk`.
- Même en connaissant très peu de choses dessus, il est déjà très utile, mais n'hésitez pas à en apprendre plus si vous aimez ce joli petit langage.



Merci à Julia Evans qui nous a gentiment laissés utiliser sa bande dessinée pour ce cours.
`https://wizardzines.com/`

BRE et ERE

POSIX définit deux langages d'expressions rationnelles (oui... c'est un de trop) :

- les *Basic Regular Expresssions* (BRE) qui, en plus des caractères déjà vus ont,
  - `\(` et `\)` pour *grouper* les caractères
  - `\{` et `\}` pour donner un intervalle pour les répétitions
- les *Extended Regular Expresssions* (ERE) qui, en plus des caractères déjà vus ont, pour *grouper* les caractères
  - `+` pour un motif répété 1, 2, ... fois
  - `?` pour un motif optionnel (0 ou 1 fois)
  - `(` et `)` pour *grouper* les caractères
  - `{` et `}` pour donner un intervalle pour les répétitions
  - `|` pour un « ou » entre deux expressions

BRE et ERE

- Sans option, `grep` et `sed` utilisent les BRE, tandis que `awk` utilise les ERE.
- `grep -E` et `sed -E` pour utiliser les ERE.