

Les expressions régulières, Sed, (Perl, Awk)

— Ensimag 1A

Grégory MOUNIÉ *



2024-2025

La grande force des outils classiques d'Unix est leur capacité de manipulation des textes avec précision, souplesse, et à grande échelle. Ce TP vise à vous familiariser avec quelques concepts et manipulations de base qui reviennent le plus souvent.

Ce TP est construit comme un tutoriel. Les commandes données en exemple sont là pour que vous les reproduisiez, les corrigiez, et jouiez avec pendant la séance. La seule lecture du sujet n'a que peu d'intérêt. Le copier-coller des commandes n'apporte pas grand-chose non plus. Alors que les retaper implique de réfléchir, de comprendre vos erreurs de retranscriptions, et permet de mémoriser les concepts en essayant des variations.

Le TP est calibré pour occuper les plus rapides d'entre-vous jusqu'à la fin du TP. Les explications abondantes sont là pour permettre aux autres de pouvoir le terminer en autonomie.

1. Outils de bases de la manipulation de texte

Un shell UNIX fournit deux points essentiels :

- un contrôle fin et facile des processus et des fichiers, que nous avons travaillé dans la séance sur les scripts

*Et les contributions de Frédéric PÉTROT, notamment sur l'usage de vim

- il est accompagné d'un très grand nombre d'outils de manipulation de texte qui ont en commun les *expressions régulières*, que nous allons explorer dans cette séance. Le but est d'explorer quelques points forts de certains des outils parmi les plus connus.

2. Les expressions régulières (ou *rationnelles*)

Une expression régulière est une séquence de caractère qui décrit un ensemble de séquences de caractères. Le plus simple exemple est la séquence "abc" qui correspond à la séquence "abc".

Les expressions régulières utilisent certains caractères pour désigner des ensembles de caractères :

- . (le point), n'importe quel caractère : `.bc` désigne `abc`, `bbc`, `cbc`, etc.
- `[]` (les crochets) désigne un ensemble (ou un intervalle) de lettres possibles (ou impossibles avec un chapeau en début) : `[ab]bc` désigne `abc` et `bbc` ; `[^ab]bc` désigne `cbc`, `dbc`, `ebc`, etc. mais pas `abc` et `bbc` ; `[a-c]bc` désigne `abc`, `bbc`, `cbc`
- `|` (le pipe), une alternative : `a|b` désigne `a` ou `b`
- `?`, 0 ou une fois : `abc?` désigne `abc` et `ab`
- `*`, 0, une ou plusieurs fois : `a*bc` désigne `bc`, `abc`, `aabc`, `aaabc`, `aaaabc`, etc.
- `+`, une ou plusieurs fois : `a+bc` désigne `abc`, `aabc`, `aaabc`, etc.
- `()` (les parenthèses) groupe une séquence, cela s'applique en combinaison avec les opérateurs précédents : `(ab)+c` désigne `abc`, `ababc`, `abababc` etc.
- `^` (le chapeau), désigne le début d'une ligne
- `$` (le dollar), désigne la fin d'une ligne

Mais comment reconnaître les caractères `+`, `|`, `*`, `?`, etc. s'ils font parti du langage à reconnaître ?

Pour différencier le symbole "+" dans une chaîne, du symbole "+" de l'opérateur *one-or-more*, les moteurs de reconnaissance d'expressions régulières utilisent le symbole `\`, ce qui donne `\+` pour l'un des deux cas, mais hélas, pas toujours le même et pas toujours pour tous cas du même moteur, en particulier pour les crochets. Pas de secret, il faut lire la documentation car votre bulle Google, ou ChatGPT, pourrait facilement vous tromper.

Question 1 *Lancer Emacs et créer un nouveau fichier (oui, oui, cet éditeur, pas un autre) `emacs lafontaine.txt` et recopier les vers suivants :*

```
Rien ne sert de courir ; il faut partir à point :
Le lièvre et la tortue en sont un témoignage.
```

Nous allons lancer l'outil interactif de construction d'expression régulière d'Emacs en faisant `M-x re-builder` (Appuyer sur la touche `Alt` et `x` (la touche `x`) simultanément, ou `Echap` et `x` (la touche `x`) successivement, puis taper le nom de la fonction `re-builder`).

Par défaut, il utilise la syntaxe dite de lecture de Emacs "read" qui impose de doubler tous les `\`. Nous allons utiliser la syntaxe "string", en indiquant "string" dans le menu "Re-builder" \Rightarrow "Change syntax...". Le raccourci du menu est indiqué "C-c TAB" : appuyer simultanément sur la touche "Control" et "c" (comme pour tuer un processus dans un

terminal) puis après les avoir relâché sur "Tabulation" (la touche la plus à gauche de la 2ème ligne, votre meilleure amie dans un terminal).

Vous devriez voir en bas de l'éditeur deux doubles quotes "" pour y insérer vos expressions régulières et voir s'afficher en haut les séquences trouvées.

Pour ceux qui n'auraient que 10 doigts, vim, ou nvim est une option intéressante. Pour rechercher une expression rationnelle¹, il suffit de taper /, et vous pouvez faire les questions qui suivent également ! En Emacs, l'équivalent correspond à faire "Edit" ⇒ "Search" ⇒ "Incremental search" ⇒ "Forward regexp (C-M-s)", mais ces recherches interactives sont un peu moins pratique d'avis d'Emacsien pour apprendre à jouer avec les expressions régulières.

Question 2 *Saurez-vous trouver les expressions régulières pour :*

1. la lettre "o";
2. la lettre "r" suivie de la lettre "i";
3. en début de ligne, la lettre "r" suivie de la lettre "i";
4. la lettre "e", suivi de la lettre "r" ou de la lettre "t";
5. la lettre "o", un nombre quelconque de lettres quelconques, puis la lettre "i" sur la même ligne. Ici, l'expression régulière maximise la taille, il est également possible de minimiser la taille. Dans les regexps étendues, cela consiste à utiliser l'opérateur "*" au lieu de "*";
6. une ligne terminant par ":" (deux points);
7. "lievre" ou "tortue" : facile, en théorie, mais difficile en pratique si vous avez oublié de changer le mode `re-builder` de "read" vers "string". Cet exemple est juste pour vous faire comprendre que parfois, exprimer une expression régulière particulière peut dépendre de son environnement (shell, python, etc.).

Concernant la dernière question, on notera que la commande `echo "lievre ou tortue" | grep --color 'lievre|tortue'` tapée dans un terminal cherche effectivement le mot lievre ou tortue. Le caractère | seul serait sinon traité comme un caractère littéral par `grep` qui gère par défaut des BREs² (Basic Regular Expressions). Lors de l'utilisation d'un shell, il est fortement conseillé d'utiliser des simples quotes, pour que le shell ne change pas la chaîne de caractère que vous tapez (interprétation des slashes, des dollars, des accolades, etc.).

Question 3 *En reprenant les recherches de la question précédente, chercher avec `grep --color`, les différentes lignes qui correspondent dans le fichier `lafontaine.txt`.*

3. Sed

`sed` est l'un des outils UNIX les plus connus. C'est un éditeur de texte en flot (*stream editor*). Il partage les mêmes commandes que `ed` (éditeur ligne à ligne) l'ancêtre direct de

-
1. Pour l'aide sur les expressions rationnelles, tapez `:help regex` sous vim.
 2. plus de détails sur <https://www.regular-expressions.info/posix.html>

`vi` (prononcer vee-ail), l'ancêtre de `vim`. Les utilisateurs de `vim` reconnaîtront donc leurs commandes habituelles.

Le concept de « traitement de données en flot » consiste à traiter chaque ligne ou bloc de données indépendamment, sans avoir besoin de charger en mémoire l'ensemble des données. Une fois le bloc de données traité, il peut être effacé de la mémoire et le bloc suivant peut être chargé.

L'avantage principal est de s'affranchir des limites de taille, qu'il s'agisse de limites liées au logiciel (par exemple 65535 lignes ou colonnes dans certains tableurs), des limites de taille mémoire (on peut traiter des données de taille gigantesque qui ne tiendraient pas dans la mémoire de votre machine), ou de limites liées à l'accès à toutes les données simultanément (pas besoin de décompresser tout le fichier d'un coup, on peut décompresser juste le bout intéressant). Par ailleurs, certaines données sont naturellement disponibles sous forme de flot, potentiellement infini : fil de tweets contenant un *hashtag* donné, vidéo en temps réel, données d'observation d'un satellite... Dans ce cas, les outils de traitement en flot sont souvent les plus adaptés.

3.1. LA commande de substitution : `s`

C'est la commande essentielle et la plus connue. Elle substitue une séquence reconnue par une expression régulière par une autre séquence.

Étape 1 Afficher "après" lorsque "avant" est lu

```
1 echo avant
2 echo avant | sed 's/avant/après/'
```

Les délimiteurs traditionnels sont le `/`, mais comme ils servent aussi pour les répertoires et les fichiers, il est possible de les remplacer par un autre caractère, à choisir librement.

Étape 2 Afficher "après" lorsque "avant" est lu en utilisant `_` (underscore) comme délimiteur.

Question 4 Que se passe-t-il dans les substitutions si l'on répète trois fois "avant" sur la même ligne ?

```
1 echo avant avant avant
```

Et sur trois lignes ?

```
1 printf "avant avant avant\navant avant avant\navant avant avant\n"
```

Il est possible de mettre des options juste après le dernier délimiteur pour changer le comportement :

Étape 3 En reprenant l'affichage des 9 "avant", ajouter successivement les options `"1"`, `"2"`, `"3"` et `"g"`.

Par exemple, pour l'option `"2"` :

```
1 printf "avant avant avant avant avant avant avant avant\n" | sed
  ↪ 's/avant/après/2'
```

3.2. Manipulation du fichier de données

On peut lire les 10 premières lignes du fichier de données sans le décompresser en entier :

```
1 zcat mydata.csv.gz | head
```

ou les 10 dernières, mais dans ce cas on va être obligé de décompresser et lire le fichier en entier :

```
1 zcat mydata.csv.gz | tail
```

Dans le reste des exercices, inutile de sauvegarder les données résultats. Il suffira souvent de n'afficher que les 10 premières lignes pour vérifier le résultat.

3.3. &, les parenthèses et \1

Nous allons manipuler le fichier de données. Le caractère & permet de reprendre dans la chaîne de remplacement, la chaîne originale.

Étape 4 En utilisant `sed` et `&`, nous pouvons ajouter la chaîne "3MMUNIX " dans la première colonne juste avant `gr[0-4]`.

Les parenthèses permettent de grouper des morceaux de séquence. Elles peuvent servir pour les opérateurs (comme `*+?`). Ces sous-groupes ont un numéro qui peut être repris dans la chaîne remplacement : `\1` sera le premier groupe, `\2` sera le deuxième groupe, etc.

Question 5 Changer l'ordre des colonnes : au début la deuxième colonne, en deuxième la troisième et en dernier la première. Le résultat final devrait ressembler à :

```
0; 13; gr4
1; 13; gr2
2; 1; gr3
...
```

Vous pourriez avoir besoin de mettre des `\` devant les parenthèses et le `+` ou d'utiliser `sed -E`.

3.4. Variante bonus : les intervalles

Il est possible de ne réaliser les transformations que dans un intervalle de lignes donné. Pour cela, on peut mettre une expression devant le "s" pour désigner la ligne ou bien un numéro, et même combiner les deux comme "1,stop s/avant/après/".

Question 6 (Difficile) Réaliser en `sed` les questions `awk` de la section B.

A. Perl

Perl est un langage de programmation généraliste, comme Python et Ruby, mais avec des opérateurs de manipulations de texte très concis. La programmation en Perl fournit, comme dans une langue naturelle, de nombreuses façons d’exprimer la même opération. Python et Ruby sont plus récents et se sont en partie construits avec, ou contre, les choix de Perl. Pour être complet, nous ne verrons ici que Perl 5. Il existe aussi Raku (ex Perl 6, disponible depuis Noël 2015), qui est le même langage de base, un peu révisé, mais enrichi de beaucoup, beaucoup plus de syntaxes et opérateurs objets, fonctionnels, grammaticaux, parallèles, concurrents, etc.

Perl est encore dominant comme langage de script dans l’administration des systèmes d’exploitations Linux. Du coup, il est installé par défaut et toujours disponible, même sur les machines les plus insolites, contrairement à Python et Ruby. Pour obtenir un comportement stable et rétro-compatible, Perl force le programmeur qui veut utiliser une fonctionnalité apparue tard, à préciser son niveau de version du langage³. Cela explique une partie de son exceptionnel longévité.

Une différence « philosophique » essentielle entre Perl et Python est visible dans l’usage de l’opérateur `.` (un point) qui sert à la concaténation des chaînes de caractères : les variables en Perl n’ont pas vraiment de types comme en python ! Mais les arguments des opérateurs de perl ont un type ! Les données sont automatiquement converties vers ce type. Par exemple, en Perl, les lignes suivantes produisent le même résultat :

```
1 1 . 1      # "11"
2 "1" . 1    # "11"
3 1 . "1"    # "11"
4 "1" . "1"  # "11"
```

produisent toute la chaîne de caractère "11" et, `"11"+"1"` vaut 12. En Python, `11+1` vaut 12, mais `"11"+"1"` vaut "111".

Étape 5 Comparer le résultat de

```
1 perl -e 'use v5.10; ; $a=11; $b=1; say $a . $b;' # concatenation
2 perl -e 'use v5.10; $a="11"; $b="1"; say $a + $b;' # addition
3 python3 -c 'a="11"; b="1"; print(a+b);' # que fait + ?
4 python3 -c 'a=11; b=1; print(a+b);' # que fait + ?
5 ruby -e 'puts 1 + 1'
6 ruby -e 'puts "1" + "1"'
7 ruby -e 'puts 1 + "1"' # produit une erreur
8 raku -e 'say "1" ~ 1' # descendant de Perl. Pour éviter la discussion,
   ↪ ils ont changé l'opérateur de concaténation
```

Vous pouvez remercier les créateurs de ces langages, Larry Wall, Guido van Rossum, Yukihiro Matsumoto et Larry Wall⁴ ...

3. dans la suite du sujet “say” qui apparaît à la version 5.10

4. Il est là deux fois : Perl et Raku

Nous allons nous servir de Perl pour générer un fichier CSV de 100000 lignes. Chaque ligne a la forme suivante : trois valeurs séparées par un point-virgule et un espace, un nom de groupe tiré au hasard (gr0 à gr4), un numéro qui s'incrémente et un entier tiré entre 0 et 20. Cela donnera quelque chose comme :

```
gr4; 0; 13
gr2; 1; 13
gr3; 2; 1
...
```

Pour l'affichage, nous utiliserons la commande `say`, qui n'existe que depuis Perl 5.10.

Étape 6 *Utiliser Perl pour afficher "Hello" concaténé à " world!" avec l'opérateur `.` et `say`.*

Pour joindre chaque élément d'une liste (entre parenthèses) avec des points-virgules et un espace, nous utiliserons la fonction `join`. Son premier argument est le séparateur, le deuxième sera un tableau, par exemple `(1, 2, 3)`

Étape 7 *Utiliser Perl pour afficher "1; 2; 3". Pour afficher le résultat de `join`, il suffit de faire `say join ...separateur..., ...tableau...` ;*

Pour tirer au hasard, nous utiliserons la commande `rand` suivi de son argument. Cette commande tire un nombre flottant de manière uniforme sur l'intervalle entre 0 et l'argument. La conversion en entier est assuré par la commande `int`.

Étape 8 *Tirer au hasard, soit 0, soit 1, uniformément et afficher la valeur. Boucler 10 fois en ajoutant `for 0..9` à la **fin** de la commande.*

Enfin, il suffit de boucler, et pour cela, comme il y a une seule commande, la boucle peut être mise en fin de ligne et elle positionne la variable implicite `$_` qui correspond à l'index de la boucle `for`.

Étape 9 *Pour vérifier que votre syntaxe est la bonne, inutile de générer toutes les lignes*

```
1 use v5.10; say join "; ", ("gr" . int rand 5, $_, int rand 21) for
  → 0..10;
```

Cette syntaxe n'est pas la seule et une version impérative verbeuse, proche des langages C et Java, pourrait être :

```
1 use v5.10;
2 for(my $i=0; $i <= 10; $i++) {
3     my $groupe = "gr" . int rand 5;
4     my $note = int rand 21;
5     say $groupe . "; " . $i . "; " . $note;
6 }
```

Étape 10 Pour produire notre jeu de données compressé pour ne pas prendre trop de place disque inutilement, il suffit de faire :

```
1 perl -e 'use v5.10; say join "; ", ("gr" . int rand 5, $_, int rand  
  ↪ 21) for 0..100000;' | gzip -9 > mydata.csv.gz
```

Il serait possible de faire, efficacement, toutes les autres manipulations de ce TP en Perl, mais il faudrait alors une véritable présentation du langage.

B. Awk

Ce langage de programmation est utilisé pour la manipulation un peu plus complexe de texte. Il combine une syntaxe assez classique (avec des if, des variables, les opérateurs arithmétiques, etc.) et des opérateurs dédiés aux flots de texte.

Pour vous expliquer les bases, nous allons utiliser `awk` pour répondre à la question suivante :

Question 7 Quelle est la taille totale prise par tous les fichiers Python `.py` dans un répertoire donné ?

Étape 11 La commande `ls` donne toutes les informations utiles : nom et taille

```
1 ls -l
```

Étape 12 Les informations pertinentes sont en colonne 5 et 9 pour les lignes contenant la chaîne `.py`.

```
1 ls -l | awk -- '/\.py/ {print $5, $9;}'
```

Attention à l'usage des simples et doubles quotes !

En pratique la sélection précédente est un peu large. Elle comptera par exemple un fichier `toto.pytoto`. On peut faire un peu mieux, en regardant spécifiquement la colonne 9 avec le nom et en vérifiant que `.py` est bien en fin de colonne.

```
1 ls -l | awk -- '$9 ~ /\.py$/ {print $5, $9;}'
```

Pour terminer il reste à :

- faire la somme des tailles dans une variable
- afficher la somme à la fin. Il existe des *matching* spéciaux : `BEGIN` et `END` qui correspondent au début et à la fin.

Étape 13 On utilise ici le *matching* de fin, `END`

```
1 ls -l | awk '$9 ~ /\.py$/ {print $5, $9; som += $5; nb++} END { print  
  ↪ "total (KiB): " som/1024 " KiB en ",nb,"fichiers parmi", NR,"  
  ↪ fichiers au total"}'
```

Note sur l’affichage Lors d’un affichage avec `print`, les arguments séparés par des virgules sont affichés dans des champs différents, avec le séparateur courant.

```
1 print $2 $1 # concatène en les inversant les deux champs
2 print $2, $1 # inverse les deux champs
```

B.1. Variante bonus : traitement correct des espaces dans les noms de fichiers

S’il y a des espaces dans le nom d’un fichier, il comptera pour 2 colonnes, ou plus. Le suffixe `.py` sera du coup en colonne 10 ou plus.

Question 8 (Difficile) *Corriger ce problème.*

B.2. Manipulation du fichier de données

Question 9 *Calculer la moyenne de la colonne 3 de notre fichier CSV à l’aide d’awk.*

Le *matching* peut contenir deux termes qui décrivent le début et la fin du *matching*. Le premier active un booléen, le deuxième le termine. Cela peut correspondre à des numéros de lignes (absence correspondant à 0 ou à la fin du fichier), ou bien à des expressions régulières. Attention, le *matching* peut s’activer plusieurs fois.

Un premier exemple, avec toutes les commandes commençant par "a" ou "b", et la première commande commençant par "c".

```
1 ls /usr/bin | awk '/^[aA]/, /^[cC]/ { print }'
```

Un deuxième exemple, pour afficher avec `ls -la` les fichiers du répertoire courant compris entre "" et ".bashrc", inclus, c’est-à-dire les fichiers et répertoire dans l’ordre alphabétique jusqu’à BASHRC (sans prendre en compte les . et les majuscules/minuscules)

```
1 ls -la ~/ | awk '$9~/^\.$/, $9~/^.bashrc$/ { print }'
```

ou bien de la 3ème ligne à la 7ème ligne (NR : number of record, le numéro de la ligne)

```
1 ls -la ~ | awk 'NR==3, NR==7 { print }'
```

Question 10 *Calculer la moyenne de la colonne 3 des numéros, en colonne 2, entre 100 et 1000.*

Question 11 *Calculer la moyenne par groupe. Il n’y a que 5 groupes, donc une méthode “brute force” fonctionne, mais vous pouvez faire plus court.*

B.3. Tableau associatif (table de hachage)

En utilisant les tables de hachage, il est facile de compter le nombre de membres dans chaque groupe :

```
1 { gr[$1]++; }  
2 END { for (i in gr) { print i, gr[i] } }
```

On notera que chaque case de la table est initialisée à zéro, ce qui permet la concision de la première ligne du script.

Question 12 *Compter et afficher la moyenne de chaque groupe.*

B.4. Variante bonus : combien d'heures de 3MMUNIX et avec qui ?

En utilisant le fichier ICS tiré de EDT pour votre emploi du temps, compter les heures de 3MMUNIX avec vos différents enseignants (certains groupes ont eu plusieurs intervenants).