

# TP de Calculabilité et Complexité (C&C)

En fait « Théorie des langages » ou « Analyse syntaxique »

Les exercices sont supposés être fait avec les outils proposés (Python, grep/Select-String/findstr) mais beaucoup d'autres outils et langages de programmation effectuent les mêmes tâches. Par exemple les SGBDs proposent la recherche (**select**) dans les tables par **pattern** (expressions régulières).

Pour tester la syntaxe des expressions régulières, amusez-vous un peu avec :

<https://www.lacl.fr/julien.grange/Enseignements/Automotus/index.html>  
!!!!!!! <https://regexcrossword.com/> !!!!!!!!! (très marrant comme site!)

## Partie A : Grep/Select-String/findstr

### Généralités

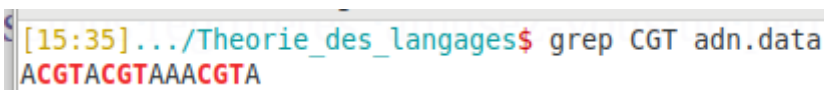
« grep » est un outil (de base sous linux/bash) permettant de retrouver une chaîne de caractères dans des fichiers (donc de trouver des **occurrences** d'un **pattern**). La version pour Windows est « findstr » (via commande MsDos) et « Select-String » (via PowerShell). Quelques liens :

<https://www.it-connect.fr/powershell-grep-rechercher-des-chaines-de-caracteres-avec-select-string/>

<https://blog.securiteinfo.com/blog/grep-pour-windows>

<https://tuteurs.ens.fr/unix/chercher.html>

<https://www.editions-eni.fr/livre/expressions-regulieres-syntaxe-et-mise-en-oeuvre-avec-exercices-et-corriges-9782746097124/extrait-du-livre.pdf>

Exemple :  


```
[15:35].../Theorie_des_langages$ grep CGT adn.data
ACGTACGTAAACGTA
```

La **concaténation** des caractères est implicite. Il est recommandé de mettre la recherche entre des guillemets 'CGT' ou "CGT" pour que l'interprétation des caractères spéciaux soit bien celle de « grep » et non du shell. Il est possible (grand classique) d'appliquer le filtre sur tout les fichiers d'un dossier et ceux récursivement :

```
grep -r CGT *.*
```

D'ailleurs, l'« \* » sous les shell est celle des expressions régulières indiquant 0...n caractères quelconques. Des options sont proposés comme par exemple « -n » pour indiquer le numéro de ligne

```
[15:39].../Theorie_des_langages$ grep -n CGT adn.data
1:ACGTACGTAAACGTA
3:AAAAACGTAAAA
```

Bien sûr quand il y a énormément d'occurrence, on peut faire défiler les messages :

```
grep -cnr import *.java | less
```

Une autre commande pratique est « find » qui fonctionne comme « grep » mais sur les noms des fichiers (très pratique si vous avez perdu un fichier sur votre disque et que vous ne savez plus trop son nom).

## Expressions régulières avec « grep »

**^** correspond au début de ligne

**\$** correspond au caractère de fin de ligne

Exemple : rechercher toutes les lignes commençant par A :

```
[15:59].../Theorie_des_langages$ grep -n '^A' adn.data
1:ACGTACGTAAACGTA
3:AAAACGTAAAA
5:AAAAAGGGGCGTTTTTTT
6:AAACGACGACGAGTTTCG
7:AAAAAAAAGUAAAAAAA
```

**[A-N]** correspond à tout les caractères compris (ordre ASCII) entre A et N. Par exemple, un chiffre serait [0-9]. On parle de plage de caractères permis.

**[^ABC]** correspond a tout sauf les caractères A, B et C (plage de caractères interdits)

**.** désigne n'importe quel caractère

**A?** correspond à le caractère A, 0 ou 1 fois (le A précède le ?)

Exemple, rechercher GTA avec un ce A optionnel

```
[16:19].../Theorie_des_langages$ grep -n 'GTA\?' adn.data
1:ACGTACGTAAACGTA
3:AAAACGTAAAA
5:AAAAAGGGGCGTTTTTTT
6:AAACGACGACGAGTTTCG
```

Pour que grep interprète littéralement ces caractères, et ne les considère plus comme spéciaux, il faut les faire précéder d'un backslash (\).

**[:alpha:]** correspond aux lettres (majuscules et minuscules mais pas les accents)

**[:space:]** correspond aux espaces (tabulation, espace, saut de ligne)

**|** correspond au « OU ». Exemple, rechercher tout les CGT ou les GA

```
[16:27].../Theorie_des_langages$ grep -n 'GA\|CGT' adn.data
1:ACGTACGTAAACGTA
3:AAAACGTAAAA
5:AAAAAGGGGCGTTTTTTT
6:AAACGACGACGAGTTTCG
```

**x\\*** zéro ou plus occurrences du caractère x ; l'opérateur est « gourmand », il prend la plus longue séquence possible

**x\+** une ou plus occurrences du caractère x ; idem « gourmand »

**\{n\}** n fois le caractère qui précède. Exemple, rechercher 3 A de suite :

```
[16:27].../Theorie_des_langages$ grep -n 'A\{3\}' adn.data
1:ACGTACGTAAACGTA
3:AAAACGTAAAA
4:TTTTAAACCCCGGG
5:AAAAGGGGCGTTTTTTT
6:AAACGACGACGAGTTTCG
7:AAAAAAAAGUAAAAAAA
```

**(EXPRESSION)** permet de construire une sous-expression régulière (appelée **groupe**), donc a grouper des expressions. Exemple, rechercher tout les lignes finissant par 3 A ou 3 T :

```
[16:42].../Theorie_des_langages$ grep -n '\(A\{3\}$\)\|\(T\{3\}$\)' adn.data
3:AAAACGTAAAA
5:AAAAAGGGGCGTTTTTTT
7:AAAAAAAAGUAAAAAAA
```

Et si vous en avez marre de mettre des \ partout, utilisez « egrep » :

```
[16:47].../Theorie_des_langages$ egrep -n '(A{3}$)|(T{3}$)' adn.data
3:AAAACGTAAAA
5:AAAAAGGGGCGTTTTTTT
7:AAAAAAAAGUAAAAAAA
```

**Remarque** : on aurait pu aussi écrire `egrep -n '((A{3})|(T{3}))$' adn.data`

Toutes les options décrites ici : <http://manpagesfr.free.fr/man/man1/grep.1.html>

**Remarque** : egrep permet de faire un peu plus que les expressions régulières avec une légère forme de répétition : si on a un groupe (une expression entre ()) alors sa valeur est « enregistrée » (*regroupé*) dans une variable qui peut être réutilisée (*référence*) avec \1 pour la première expression entre () puis \2 pour la seconde etc. Exemple, un palindrome de 3 caractères peut s'écrire : `(\w)\w\1`. Pour 5 caractères `(\w)(\w)\w\2\1`. On a utilisé alors les raccourcis suivants :

Raccourci	Signification
<code>\d</code>	Indique un chiffre. revient exactement à taper [0-9]
<code>\D</code>	Indique ce qui n'est PAS un chiffre. revient à taper [^0-9]
<code>\w</code>	Indique un caractère alphanumérique ou un tiret de soulignement. Cela correspond à taper [a-zA-Z0-9_]
<code>\W</code>	Indique ce qui n'est PAS un caractère alphanumérique ou un tiret de soulignement. revient à taper [^a-zA-Z0-9_]
<code>\t</code>	Indique une tabulation
<code>\n</code>	Indique une nouvelle ligne
<code>\r</code>	Indique un retour chariot
<code>\s</code>	Indique un espace blanc (correspond à \t \n \r)
<code>\S</code>	Indique ce qui n'est PAS un espace blanc (\t \n \r)
<code>.</code>	Le point indique n'importe quel caractère ! Il autorise donc tout !

**Remarque** : pour finir, voici une expression **Perl** regex pour des palindromes de n'importe quel taille (impossible avec grep car ici Perl autorise l'utilisation de \2 avant même que l'expression soit évaluée, bref, c'est récursif...) :  `/^(.)(?1)\2(?:.?)$/`

## Exercice 1

Recherchez dans le fichier adn.data :

- Toutes les lignes ne finissant pas par un caractère de fin de phrase ; . ! ? :
- Toutes les lignes où figurent plusieurs A de suite. Plusieurs veut dire au moins 2.
- Toutes les lignes où plusieurs A ne terminent pas la ligne.
- Les lignes où on trouve C puis G puis T dans l'ordre avec n'importe quel caractère entre ces 3 caractères

## Exercice 2

Pour un fichier quelconque, donnez une expression régulière qui liste tout les mots contenant une lettre doublée. Un mot est une suite de caractères avec soit un (ou des) espace devant soit le début de ligne. Exemple : `egrep '^.[[:space:]]|.r' fichier` permet de trouver tout les mots dont la seconde lettre est un « r ». Indice : il faudra utiliser un regroupement et une référence...

Ensuite, complétez avec une expression régulière qui donne les mots avec lettre doublée qui ne sont pas des noms propres. (C'est à dire qui ne commencent pas par une lettre majuscule...)

## Exercice 3

Le fichier `/etc/passwd` contient les utilisateurs d'un Unix mais aussi de logiciels à mot-de-passe comme MySQL. Le fichier contient par ligne, le login suivi de « : » suivi d'autres informations. Exemple :

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...
gava:x:1000:1000:gava,,,:/home/gava:/bin/bash
mysql:x:125:134:MySQL Server,,,:/nonexistent:/bin/false
```

Comment rechercher dans ce fichier tous les utilisateurs dont le nom contient un chiffre ?

# Partie B : Le module « re » en Python

## Généralités

Pour utiliser les expressions régulières sous Python il faut importer le module « re ». Exemple d'utilisation simple (fichier **testExpr.py** sous Teams) :

```
import re
def main():
    findall = re.findall("EPISEN", "L'EPISEN est la meilleure école, vive l'EPISEN")
    print(findall) // rechercher toute les occurences
    search = re.search("EPISEN", "L'EPISEN est la meilleure école, vive l'EPISEN")
    print(search) // rechercher la première
    findall = re.findall("[a-rt-z]*e", "L'EPISEN est la meilleure école, vive l'EPISEN")
    print(findall)
    search = re.search("[a-rt-z]*e", "L'EPISEN est la meilleure école, vive l'EPISEN")
    print(search)
main()
```

Ce qui donne :

```
[15:21].../Theorie_des_langages$ python3.12 testExpr.py
['EPISEN', 'EPISEN']
<re.Match object; span=(2, 8), match='EPISEN'>
['e', 'meilleure', 'cole', 'vive']
<re.Match object; span=(9, 10), match='e'>
[15:21].../Theorie_des_langages$ █
```

Il y a bien 4 fois une série (même vide pour le « est ») de lettres avant un « e ».

Il est aussi possible de compiler une expression régulière ; ensuite le résultat est stocké dans un objet RegexObject pour être potentiellement utilisé plusieurs fois. Exemple (chaînes formées d'une ou plusieurs lettres entre a et z) :

```
import re
def main():
    r = re.compile(r"[a-z]+")
    findall = r.findall("les chaussettes de l'archiduchesse sont-elles sèches, archi sèches ?")
    print(findall)
```

```
[16:11].../Theorie_des_langages$ python3.12 testExpr.py
['les', 'chaussettes', 'de', 'l', 'archiduchesse', 'sont', 'elles', 's', 'ches', 'archi', 's', 'ches']
```

L'objet « r » a plusieurs méthodes, dont voici les principales (fonctionnalités similaires «chercher / remplacer» des éditeurs de texte) :

- \* **findall** qui sert à appliquer l'expression régulière et à récupérer les sous-chaînes trouvées sous forme de liste
- \* **finditer** qui sert à appliquer l'expression régulière et à récupérer les sous-chaînes trouvées sous forme d'itérateur
- \* **sub** qui sert à appliquer une expression régulière, à remplacer les sous-chaînes trouvées par d'autres chaînes.
- \* **match**, tester si une chaîne satisfait les contraintes imposées par l'expression régulière auquel on l'applique

Exemple de **finditer** :

```
r = re.compile(r"[a-rt-z]*s[a-z]*") # les mots contenant un « s »
for m in r.finditer("les chaussettes de l'archiduchesse sont-elles sèches, archi sèches ?"):
    print(m.group())
```

```
les
chaussettes
archiduchesse
sont
elles
s
ches
s
ches
```

(N.B. : les accents sont gérés comme des espaces ici)

Exemple de **sub**:

```
r = re.compile(r"s")
m = r.sub(r"ch", "les chaussettes de l'archiduchesse sont-elles sèches, archi sèches ?")
print(m)

r = re.compile(r"s+([a-z]+)")
m = r.sub(r"ch\1", "Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

lech chauchchettech de l'archiduchechche chont-ellech chèchech, archi chèchech ?  
les chauchettes de l'archiducheeche chont-elles sèches, archi sèches ?

Dans le premier « sub », nous remarquons que « s », muet, de « les » est remplacé par « ch », ce qui n'est vraiment pas valide, même pour un virelangue. Pour ce faire, on a créé un groupe ([a-z]+) que l'on retrouve dans la chaîne de remplacement (\1).

Aussi, lorsqu'on remplace une sous-chaîne par une autre il peut être utile d'intercaler un traitement entre lecture de la sous-chaîne et écriture dans la nouvelle chaîne. Python nous permet d'appliquer une fonction à chacune des sous-chaînes trouvées. Imaginons que dans la chaîne

toto 123 blabla 456 titi

et on veut représenter les nombres en hexadécimal. Ce calcul est impossible à réaliser avec uniquement des expressions régulières, on utilisera donc une fonction :

```
def ecrire_en_hexa (entree):
    return hex( int(entree.group()) )

def main() :
    r = re.compile(r"[0-9]+")
    m = r.sub( ecrire_en_hexa, "toto 123 blabla 456 titi")
    print(m)
```

```
[16:36].../Theorie_des_langages$ python3.12 testExpr.py
toto 0x7b blabla 0x1c8 titi
```

L'argument de la fonction est un objet de type **MatchObject**. La méthode **group()** fournit la chaîne tout entière, alors que **group(n)** fournira le n-ième groupe de la sous-chaîne.

## Syntaxe des expressions régulières « à la Perl »

Comme précédemment :

**toto** va trouver les sous-chaînes toto ;

**.** est un caractère quelconque, mis à part le passage à la ligne \n et le retour chariot \r ;

**[ax123Z]** signifie un caractère quelconque parmi a, x, 1, 2, 3 et Z ;

**[A-Z]** signifie un caractère quelconque dans l'intervalle de A à Z ; le trait d'union peut faire partie des caractères recherchés s'il est placé à la fin ; Exemple **[AZ-]** signifie : «un caractère quelconque parmi A, Z et -» ; on peut combiner à volonté les caractères énumérés et les intervalles : par exemple **[A-Za-z0-9.:?]**

les caractères **( , ), \, [, ]** peuvent être recherchés, à condition de les protéger par un antislash : **\(, \), \\, \[, \]** ;

**^[ ]** indique que l'on va chercher le complémentaire de ce qui est placé entre les crochets ;

on dispose des opérateurs **gourmand** (la plus longue chaîne possible) suivants :

**\*** (zéro, une ou plusieurs fois),

**+** (une ou plusieurs fois),

**?** (zéro ou une fois),

`{n,m}` (entre n et m fois),

`{n,}` (plus de n fois) ;

On obtient des opérateurs **non-gourmand** (la plus courte) en mettant un « ? » derrière : `*?` `+?` `??` `{n,m}?` `{n,}?`

Exemple, pour « Créteil (93) Paris (75) », l'expression `\(.+)` va retourner la plus longue chaîne commençant par ( et se terminant par ), soit « (93) Paris (75) », ce qui n'est pas forcément le but recherché. Par contre `\(.+?)` va bien retourner « (93) » et « (75) ».

`^` et `$` servent à indiquer respectivement le début et la fin d'une chaîne

`|` sert à indiquer un choix entre deux expressions ;

On peut délimiter une expression par des ( ) pour un opérateur comme `|` ou `*`. Exemple : `abc(toto)+` signifie «abc suivi d'un ou plusieurs toto» ; Comme avec grep, cette sous-chaîne (appelée **groupe**) peut aussi être récupérée par la suite. Si on veut éviter ce sur-coût (si on ne réutilise pas le groupe), on peut écrire `abc(?:toto)+`

**Remarque** : les expressions régulières sont rapidement cryptiques à lire avec pleins de symboles dont la signification est sémantiquement importante. Je conseil vivement de documenter les expressions pour le lecteur des codes !

## Exercice 4

Écrire une expression régulière qui reconnaisse simultanément la séquence de nucléotidiques suivante CCTCTAAAAA**TT**TATT où le nucléotide souligné gras est variable, donc potentiellement n'importe quelle lettre (ACGT). Essayez d'écrire l'expression la plus compacte possible.

## Exercice 5

Pour lire un fichier « csv » avec Python, on peut tout simplement écrire :

```
f = open("file.csv", 'r')
for ligne in f:
    #faire quelque chose avec la ligne ligne
f.close()
```

Donc, pour le fichier « data1.csv », expliquez ce que fait le code suivant ?

```
r = re.compile(r"^[0-9]+;[^;]*;Damien$")
f = open("data1.csv", 'r')
for ligne in f:
    for m in r.finditer(ligne):
        print(m.group(1) + " OK")
f.close()
```

Donc, pour le fichier « data1.csv », expliquez ce que fait le code suivant ?

## Exercice 6

Reprenez ce programme et utilisez le fichier « date2.csv » afin qu'il affiche les noms des gens nés dans une ville dont le nom commence par « Na ».

## Exercice 7

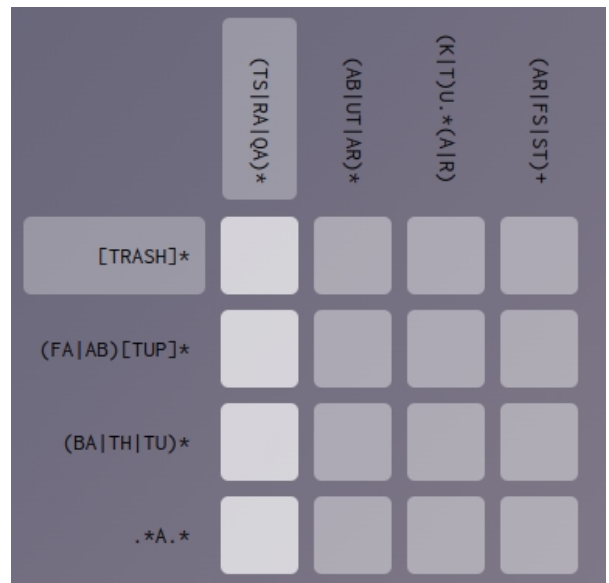
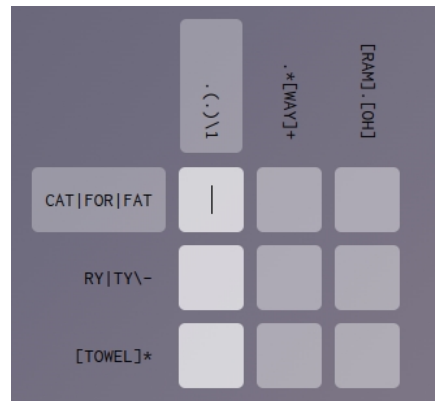
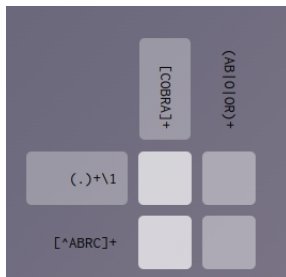
L'arginine est codée par 6 codons : CGA, CGC, CGT, CGC, AGA et AGG. Écrivez et coder une expression régulière qui reconnaisse exactement ces six codons et seulement ceux-ci. Trouvez une forme concise de cette expression.

**Remarque** : on trouve sur le web différent module/package pour Python ou Java (et dans la plus part des langages) pour la manipulation d'automates. Nous n'en utiliserons pas ici explicitement par manque de temps et aussi parce que l'intérêt est limité (à part lire les codes des algorithmes).

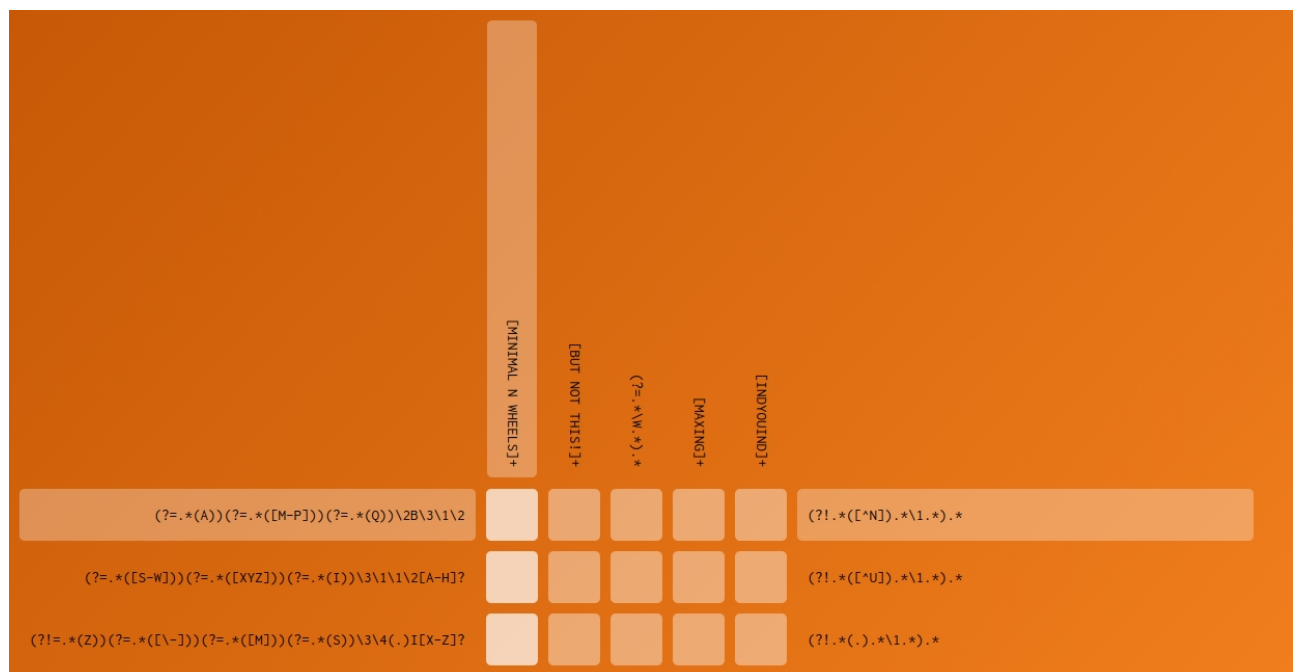
**Remarque :** ce même type de class (module) existe aussi en Java. Nous verrons à la prochaine séance un outil (écrit en Java) et qui permet de faire des analyses beaucoup plus riche (et « facilement ») de fichiers de données. Famille d'outil qui est à la base de la part des compilateurs ou outils pour l'analyse de codes.

## Exercice 8

Quelques puzzles trouvés sur <https://regexcrossword.com/>

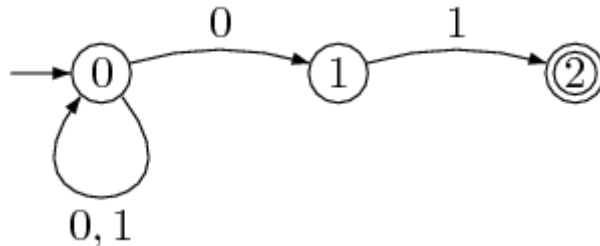


Allez bon courage ^^



## Partie C : Un petit code Java pour simuler un automate

Vous trouverez sur Teams, le code (minimaliste) **TestAutomate.java**. Celui-ci permet d'exécuter le petit automate (non-déterministe) suivant (qui accepte toutes les chaînes qui se terminent par "01") :



	0	1
→ 0	{0,1}	{0}
1	∅	{2}
* 2	∅	∅

On voit bien que cela n'est pas agréable à programmer ni efficace à exécuter même si on essaye d'optimiser manuellement. Compilation et exécution :

```

[14:28].../Theorie_des_langages$ javac TestAutomata.java
[14:29].../Theorie_des_langages$ java TestAutomata 0000000
accepter = false
[14:30].../Theorie_des_langages$ java TestAutomata 0011001
accepter = true
[14:30].../Theorie_des_langages$ java TestAutomata 0012001
accepter = false
[14:30].../Theorie_des_langages$ █
  
```

Si « grep » et « module RE Python » arrive à analyser des codes avec expressions régulières (donc de construire des automates) pour « juste » détecter ou non la présence de l'expression dans un fichier, il serait intéressant d'avoir son propre outils permettant de manipuler et programmer (en Java/Python/...) ces présences. Ce sera le rôle du TP de fin (voir précédente remarque).