Chapitre 4: Expressions régulières

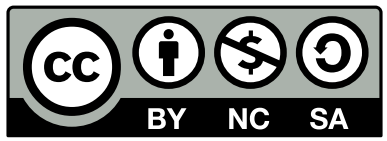
INF1070

Utilisation et administration des systèmes informatiques

Jean Privat & Alexandre Blondin Massé

Université du Québec à Montréal

v243



Plan

Introduction

Expressions régulières

- Un **motif** sous forme de **chaîne de caractères**
- Qui représente un **ensemble** (potentiellement infini) de chaînes de caractères

Objectifs

- Identifier du texte ou des parties de texte
- Chercher dans du texte
- Transformer du texte

Ça ressemble un peu au glob

Rappel: développement des noms de fichiers (glob)

Désigner simplement un ensemble de fichiers selon un **motif**

- Point d'interrogation `?` — un caractère quelconque
- Étoile `*` — zéro, un ou plusieurs caractères
- Crochets `[]` — un seul des caractères de la liste

Exemples

- « `cat *.txt` » se terminent par `.txt`
- « `cat ?[oa]*` » la deuxième lettre est `a` ou `o`

Note

- Le glob est géré par le shell (pas par la commande)
- La commande *ne voit* que les arguments une fois développés

Les détails: le manuel de `glob`

Développement des noms de fichiers (suite)

Autres motifs permis

- Point d'exclamation `[!...]` — caractère qui **n'est pas** dans la liste
- Intervalles `[<debut>-<fin>]` — caractère entre `<debut>` et `<fin>`

Exemple

- « `ls *[!co]` » — ne terminent ni par `c` ni par `o`
- « `ls [f-i]*` » — commencent par `f`, `g`, `h` ou `i`

Entre crochets, les caractères deviennent littéraux

- « `cat []*[!?*]*` » — commence par `]`, `[`, `!`, `?` ou `*`

Glob $\neq$ expression régulière

Différences

- *glob* $\rightarrow$ noms de fichier
- expressions régulières $\rightarrow$ texte
- Les conventions sont différentes

Extrait du manuel de glob

> *« Note that wildcard patterns are not regular expressions, although they are a bit similar. First of all, they match filenames, rather than text, and secondly, the conventions are not the same: for example, in a regular expression '*' means zero or more copies of the preceding thing. »*

Qui utilise les expressions régulières ?

Outils de manipulation de texte

- Éditeurs de texte: vi, et tous les éditeurs modernes, ...
- Outils autonomes: grep, sed, ...
- Base de données: MySQL, MariaDB, Oracle, ...

Langages de programmation

- De base en Perl, Python, Java, JavaScript, etc.

$\rightarrow$ Opérations sur le texte

$\rightarrow$ Vérification et sanitarisation de données

Autres utilisations

- Colorateur syntaxique colorie les mots clés
- Serveur web filtre et transforme des requêtes HTTP
- Compilateur reconnaît les éléments textuels d'un programme

Différentes conventions

De **nombreux** outils et usages

Font que la syntaxe et les mécanismes **varient**

- POSIX *Basic regular expression* (BRE)
- POSIX *Extended regular expression* (ERE)
- Extensions GNU à POSIX
- *Perl compatible regular expression* (PCRE)
- Spécificités de chaque langage et outil

Exemple de différences

- Échappement des caractères spéciaux
- Classes de caractères
- Mécanismes plus avancés (*lookahead*, références relatives, etc.)

Dans le cours: **BRE** et **ERE** (POSIX)

Plusieurs commandes les utilisent, notamment `grep` et `sed`

La commande grep: rappel

`grep` — cherche les lignes correspondant à un motif

- `-i`, `--ignore-case` ignorer la casse (majuscule/minuscule)
- `-n`, `--line-number` afficher le numéro de ligne
- `-v`, `--invert-match` afficher les lignes qui ne correspondent pas
- `-x`, `--line-regexp` chercher la ligne entière
- `--color` colorier le motif (GNU)

Autres options utiles de grep

- `-o`, `--only-match` afficher seulement la correspondance (GNU)
- `-l`, `--files-with-matches` lister les fichiers qui ont des résultats
- `-L`, `--files-without-matches` lister les fichiers qui n'ont pas de résultats
- `-c`, `--count` compter le nombre de correspondances
- `-A`, `--after-context` afficher des ligne de contexte après (GNU)
- `-B`, `--before-context` afficher des ligne de contexte avant (GNU)
- `-C`, `--context` afficher des lignes de contexte autour (GNU)
- `-r`, `--recursive` chercher dans des répertoires (GNU)
- `-f`, `--file` lire les motifs depuis un fichier
- `-E`, `--extended-regexp` passer en mode étendu (on va y revenir)
- `-F`, `--fixed-strings` chercher des chaînes fixes (désactive les expressions régulières)
- `-P`, `--perl-regexp` passer en mode PCRE (GNU)

Par défaut grep cherche des expressions régulières basiques (BRE)

Questions sur grep

- Combien de mots en français contiennent « tata » ?

Questions sur grep

- Combien de mots en français contiennent « tata » ?

→ `grep -c tata /usr/share/dict/french`

- Quel mot précède « impact » dans le dictionnaire ?

Questions sur grep

- Combien de mots en français contiennent « tata » ?

→ `grep -c tata /usr/share/dict/french`

- Quel mot précède « impact » dans le dictionnaire ?

→ `grep -B 1 impact /usr/share/dict/french`

→ `grep -B 1 impact /usr/share/dict/french | head -n1`

- Quels dictionnaires ne contiennent pas le mot « immunologie » ?

Questions sur grep

- Combien de mots en français contiennent « tata » ?

→ `grep -c tata /usr/share/dict/french`

- Quel mot précède « impact » dans le dictionnaire ?

→ `grep -B 1 impact /usr/share/dict/french`

→ `grep -B 1 impact /usr/share/dict/french | head -n1`

- Quels dictionnaires ne contiennent pas le mot « immunologie » ?

→ `grep -L -r immunologie /usr/share/dict/`

Expressions régulières de base

Généralités

- BRE = *basic regular expressions*
- Une des versions syntaxiques POSIX (l'autre étant ERE)
- Reconnue par la plupart des commandes Unix
- Reconnue par plusieurs éditeurs de texte (Vi/Vim, Emacs)
- Syntaxe de base plus ou moins universelle

Caractères littéraux et caractères spéciaux

Caractères spéciaux

- Le **point** `.`
- Les **crochets** `[` et `]`
- L'**étoile** `*`
- Le **circonflexe** `^`
- Le **dollar** `$`
- La **contre-oblique** `\`
- Tous les autres caractères sont littéraux
- « `\` » rend littéral un caractère spécial et inversement

Attention

- Certains caractères sont spéciaux pour le shell
- BRE $\neq$ glob même s'il y a des caractères communs

Le point: joker

Le **point** « . » correspond à n'importe quel caractère sauf NUL

```
$ grep sa.z /usr/share/dict/french
ersatz
sanza
$ grep anti..................... /usr/share/dict/french
anticonstitutionnellement
```

Note

- Équivalent au « ? » en glob
- Utiliser « \. » pour reconnaître un point
- Certains outils ou options n'incluent pas la fin de ligne dans « . »

L'étoile: répétition

L'**étoile** * indique une répétition d'un motif

- 0 fois
- 1 fois
- ou plusieurs fois

```
$ cat fruits.txt
banane
mangue
pomme
$ grep -o 'm*e' fruits.txt
e
e
mme
```

Note

- Protéger l'étoile du shell

Ancrage

Deux caractères spéciaux permettent de forcer une **position**

- Le circonflexe « ^ » indique le début de la chaîne (ou ligne)
- Le dollar « $ » indique la fin de la chaîne (ou ligne)

```
$ grep '^fricas' /usr/share/dict/french
fricassa
fricassai
fricassaient
[...]
fricassons
$ grep 'key$' /usr/share/dict/french
hockey
jockey
$ grep '^mettre$' /usr/share/dict/french
mettre
```

- `-x` équivaut à ajouter des `^` et `$` implicites
- Protéger `$` du shell

Question

Rechercher tous les mots du français qui contiennent « `cri` » et finissent par « `al` »

Question

Rechercher tous les mots du français qui contiennent « `cri` » et finissent par « `al` »

```
$ grep 'cri.*al$' /usr/share/dict/french
cristal
microcristal
scriptural
```

Crochets : liste de caractères

« `[]` » listent un choix parmi plusieurs caractères

```
$ echo "mes aieux" | grep -o '[aeiou]*'
e
aieu
```

Note

- Protéger « `[` » et « `]` » du shell

Crochets : négation de classe

- Entre les crochets,
- On peut interdire un ou plusieurs caractères
- En commençant par le circonflexe « ^ »

```
$ echo "mes aieux" | grep -o '[^aeiou]*'
m
s
x
```

Note

- Le circonflexe ^ joue deux rôles (ancrage et négation)

Crochets : intervalles

- Entre les crochets, on peut préciser un intervalle
- Avec le caractère « - »

```
$ grep '[x-z][a-c][y-z]' /usr/share/dict/french
oxycyanure
polycyclique
zazou
[...]
zézayons
$ grep -o '[x-z][a-c][y-z]' /usr/share/dict/french | sort -u
ycy
zay
zaz
$ echo '!@#$%^&-=+()[]' | grep --color '[!-*]'
```

Les intervalles peuvent dépendre

- De la *locale* (détails plus tard)
- ou du codage (ASCII, Unicode, etc.) `man ascii`

Classes de caractères POSIX

Certains ensembles de caractères sont plus fréquents

- `[:lower:]` : minuscules
- `[:upper:]` : majuscules
- `[:digit:]` : chiffres
- `[:alpha:]` : minuscules et majuscules
- `[:alnum:]` : minuscules, majuscules et chiffres
- `[:punct:]` : `! . ; + # / \ |` etc.
- `[:blank:]` : espace et tab
- `[:space:]` : espace, tab, `\r`, `\n`, etc.
- `[:cntrl:]` : caractères de contrôle (NUL, BS, ESC, DEL, etc.)
- `[:graph:]` : caractères graphiques, `[:alnum:]` et `[:punct:]`
- `[:print:]` : caractères affichables, `[:graph:]` et espace
- `[:xdigit:]` : caractères hexadécimaux

Les classes dépendent de la *locale* (détails plus tard)

Autres classes de caractères

GNU, Perl et de nombreux outils reconnaissent d'autres classes

- `\w` : lettre, chiffre ou souligné « _ »
- `\W` : l'inverse de `\w`
- `\s` : un espace
- `\S` : l'inverse de `\s`
- `\d` : un chiffre (pas GNU)
- `\D` : l'inverse de `\d` (pas GNU)
- `\b` : frontière entre mot et non-mot
- `\B` : l'inverse de `\b`
- `\<` : début de mot
- `\>` : fin de mot

En fonction des outils, les classes dépendent (ou non) de la *locale*

Frontières

```
$ grep 'garde' /usr/share/dict/french
arrière-garde
avant-garde
entre-regarde
[...]
sous-garde

$ grep '\bgarde\b' /usr/share/dict/french
arrière-garde
avant-garde
garde
[...]
sous-garde
```

Classes d'équivalence POSIX

- `[=c=]` : classe d'équivalence du caractère `c`
- Par exemple `[=e=]` inclut `eéèêë`

```
$ grep '[[=e=]][[=e=]][[=e=]]' /usr/share/dict/french
agréée
agréées
capéée
[...]
toréées
```

Attention aux **doubles** crochets

```
 $ echo 'e + é = è' | grep -o '[=e=]'
e
=
```

Les classes d'équivalence dépendent aussi de la *locale*

Collations POSIX

- *digraphe* = 2 caractères qui en forment un seul
- Certaines langues ont des *digraphes*
- Exemple, en tchèque, `ch` est une lettre entre `h` et `i`
- En croate, `lj` est une lettre entre `l` et `m`
- Avant, en allemand, `ss` était une lettre à part (maintenant β)
- La norme POSIX permet de gérer ces doubles caractères
- Dans le cours, on ne s'en préoccupera pas (pas de problème en français/anglais)

Question

Rechercher tous les mots du français qui ne contiennent aucune voyelle parmi `a`, `e`, `i`, `o`, `u`

Question

Rechercher tous les mots du français qui ne contiennent aucune voyelle parmi `a`, `e`, `i`, `o`, `u`

- `grep '[^aeiou]' /usr/share/dict/french`

→ reconnaît `oiseau` à cause du `s`

- `grep '[^aeiou]*' /usr/share/dict/french`

→ reconnaît tout car `*` accepte 0 répétitions

- `grep '^[^aeiou]*$' /usr/share/dict/french`

→ reconnaît `âgé` à cause des accents

- `grep '^[^aâàeéèêëiîïoôuûü]*$'`

→ fonctionne (même si ù manque)

- `grep '^[^[=a=][=e=][=i=][=o=][=u=]]*$'`

→ fonctionne aussi

Avec les options de grep

- `grep -x '[^[=a=][=e=][=i=][=o=][=u=]]*'`
- `grep -v '[[=a=][=e=][=i=][=o=][=u=]]'`

La commande sed (1/3)

`sed` (*stream editor*) filtrer et transformer du texte

- `-n` mode silencieux
- `-f`, `--file` lecture des transformation à partir d'un fichier
- `-i`, `--in-place` modifie le fichier (au lieu d'afficher sur `stdout`) (GNU)
- `-E` passer en mode étendu (GNU)
- `--posix` désactive les extensions GNU (GNU)

Commandes spécifiques à sed

- `sed` lit une ou plusieurs commandes
- Et les exécutent une à une
- Par exemple, `s/ancien/nouveau/` pour substituer `ancien` par `nouveau`
- Scriptable: on peut les regrouper dans un fichier
- Nous ne verrons que quelques-unes de ces commandes

La commande sed (2/3)

Permet (entre autres) de remplacer un motif par un autre

```
$ cat form.txt
Nom: <nom>
Prénom: <prénom>
Code permanent: <code>
Courriel: <courriel>

$ sed 's/<prénom>/Alexandre/' form.txt
Nom: <nom>
Prénom: Alexandre
Code permanent: <code>
Courriel: <courriel>
```

La commande sed (3/3)

```
$ cat remplir.sed
s/<nom>/Blondin Massé/
s/<prénom>/Alexandre/
s/<code>/BLOA00000000/
s/<courriel>/blondin_masse.alexandre@uqam.ca/

$ sed -f remplir.sed form.txt
Nom: Blondin Massé
Prénom: Alexandre
Code permanent: BLOA00000000
Courriel: blondin_masse.alexandre@uqam.ca
```

Mythologie de grep

Éditeur `ed` (POSIX)

- Développé par Ken Thompson
- `g/re/p` est une commande de `ed`
- *globally search the regular expression then print the line*

Outil `grep` (POSIX)

- Développé comme un outil autonome plus rapide
- A popularisé l'utilisation (et la notation) des expressions régulières

Dans le dictionnaire: nom et verbe

- https://en.oxforddictionaries.com/definition/grep

Expressions étendues

Différences avec BRE

Syntaxe différente

- Plus de caractères spéciaux
- Plus d'opérations

Pour l'avoir

- `grep -E` (POSIX), `egrep` (pas POSIX)
- `sed -E` (GNU)
- Par défaut dans de nombreux outils

$\rightarrow$ C'est le défaut quand les gens pensent expressions régulières

Note

- Certains caractères spéciaux BRE existent sous forme échappée
- « `(i.|.i){6}` » en ERE équivaut à
- « `\(i.\|.i\)\{6\}` » en BRE

Sous-expressions

- Les parenthèses « `()` » forment des groupes
- Les opérateurs comme `*` s'appliquent sur les groupes

```
$ grep -Ex 'hu(.i)*ent' /usr/share/dict/french
huaient
huent
humidifiaient
humidifient
humiliaient
humilient
```

Motif optionnel

« `?` » rend un motif optionnel

- On peut rendre un caractère optionnel

```
$ grep -xE 'p?r?is' /usr/share/dict/french
pis
pris
ris
```

- Ou une sous-expression complète

```
$ grep -Ex 't(rav)?aill((er)?ions)?'\
> /usr/share/dict/french
taillerions
taillions
travaillerions
travaillions
```

Question: que trouve « `man?g?e?r?i?e?z?` » ?

Répétition stricte

- L'étoile « * » accepte lorsqu'il y a 0 occurrence

```
$ grep -xE '(cher)*(as)*' /usr/share/dict/french
as
cher
chercher
chercheras
```

- Le plus « + » force au moins 1 occurrence

```
$ grep -xE '(cher)+(as)+' /usr/share/dict/french
chercheras
```

Répétitions d'un nombre précis d'occurrences

À l'aide d'accolades « `{ }` »

- `{m}` : exactement `m` fois
- `{m,}` : `m` fois ou plus
- `{,n}` : `n` fois ou moins
- `{m,n}` : entre `m` et `n` fois

```
$ echo 'aaaaaabaabaaa' | grep -oE 'a{5}'
aaaaa

$ echo 'aaaaaabaabaaa' | grep -oE 'a{3,4}'
aaaa
aaa
```

Exercice

Quels mots du français ont

- un `i` une lettre sur deux
- et au moins trois `i`?

Exercice

Quels mots du français ont

- un `i` une lettre sur deux
- et au moins trois `i`?

```
$ grep -Ex '.?i(.i){2,}.?' /usr/share/dict/french
bikini
inimitié
initia
[...]
vivifié
```

Répétitions en bref

Plusieurs façons de gérer les répétitions

- `*`, `{0,}` ou `{,}` : 0, 1 ou plusieurs fois
- `?`, `{0,1}` ou `{,1}` : 0 ou 1 fois
- `+` ou `{1,}` : 1 fois ou plus
- `{m}` : exactement `m` fois
- `{m,n}` : entre `m` et `n` fois
- `{m,}` : `m` fois ou plus
- `{,n}` : `n` fois ou moins

Avarice

En POSIX, les répétitions ont un comportement **avare** (*greedy*)

- Elles consomment le plus possible de caractères qui correspondent
- Elles trouvent les répétitions les plus longues possibles

```
$ cat cobal
cobalourdeaubobardumbadaududos
$ grep -o 'b.*d' cobal
balourdeaubobardumbadaudud
```

Questions d'avarices

Seulement les chaînes qui commencent par « `b` »
et finissent par le « `d` » le plus proche?

Questions d'avarices

Seulement les chaînes qui commencent par « `b` »
et finissent par le « `d` » le plus proche?

```
$ grep -o 'b[^d]*d' cobal
balourd
bobard
bad
```

Seulement les chaînes qui commencent par « `ba` »
et finissent par le « `ud` » ou le « `rd` » le plus proche?

Questions d'avarices

Seulement les chaînes qui commencent par « `b` »
et finissent par le « `d` » le plus proche?

```
$ grep -o 'b[^d]*d' cobal
balourd
bobard
bad
```

Seulement les chaînes qui commencent par « `ba` »
et finissent par le « `ud` » ou le « `rd` » le plus proche?

```
$ grep -o 'ba[^d]*[ur]d' cobal
balourd
bard
$ grep -o 'ba[^ur]*[ur]d' cobal
bard
badaud
```

- Possible, mais demande plus d'opérateurs

Anti-avarice

PCRE permet le contrôle de l'avarice des répétitions

- Une répétition suivie d'un point d'interrogation `?`
- N'est plus avare

L'option `-P` de `grep` active le mode PCRE (GNU)

```
$ grep -oP 'b.*?d' cobal
balourd
bobard
bad

$ grep -oP 'ba.*?[ur]d' cobal
balourd
bard
badaud
```

Alternance

- Équivalent au *ou* logique ou à l'*union* ensembliste

```
$ grep -E 'comberio|ustibi' /usr/share/dict/french
combustibilité
incomberions
incombustibilité
succomberions

$ grep -E 'e(cou|pli)(sse|é)$' /usr/share/dict/french
replié
replisse
secoué
secousse
```

Priorités

Il y a une priorité sur les opérateurs

```
$ grep -Ex 'pro|ton*s?' /usr/share/dict/french
```

Priorités

Il y a une priorité sur les opérateurs

```
$ grep -Ex 'pro|ton*s?' /usr/share/dict/french
```

- « `pro|ton*s?` » vaut « `(pro)|(to(n)*(s)?)` »
- Essayer « `pro|(ton)*s?` » et « `(pro|ton)*s?` »

Problème

- Qu'en est-il du *et* logique?

Rechercher tous les mots qui contiennent au moins une fois chaque voyelle `a`, `e`, `i`, `o`, `u`.

Quel est le problème de « `a.*e.*i.*o.*u` » ?

Problème

- Qu'en est-il du *et* logique?

Rechercher tous les mots qui contiennent au moins une fois chaque voyelle `a`, `e`, `i`, `o`, `u`.

Quel est le problème de « `a.*e.*i.*o.*u` » ?

```
$ grep a.*e.*i.*o.*u /usr/share/dict/french
garde-chiourme
gardes-chiourme
gardes-chiourmes
```

Solution

Solution possible: utiliser un **tube**

```
$ grep a /usr/share/dict/french | grep e | grep i\
> | grep o | grep u
abasourdie
abasourdies
[...]
zygomatiques
```

- Pas optimal, car relit plusieurs fois chaque ligne
- Mais pas d'autres solutions avec `grep`
- Il existe des solutions plus puissantes, par exemple Perl (plus tard...)

Capture de sous-chaîne

- On peut réutiliser une ou plusieurs sous-chaîne
- Une sous-chaîne correspond à une sous-expression
- On y réfère avec « `\i` », où « `i` » est le numéro de la sous-expression

```
$ grep -Ex '(....)..\1' /usr/share/dict/french
rentrèrent
saisissais
sentissent

$ grep -Ex '(...)(...)\2\1' /usr/share/dict/french
entassassent
```

Sous-expressions imbriquées

- On peut imbriquer des sous-expressions
- Et réutiliser les sous-chaînes associées

Question

Quels mots ont 2 fois la même paire de lettres répétées ?
Exemple: « `gouttelette` » (deux fois « tt »)

Sous-expressions imbriquées

- On peut imbriquer des sous-expressions
- Et réutiliser les sous-chaînes associées

Question

Quels mots ont 2 fois la même paire de lettres répétées ?
Exemple: « `gouttelette` » (deux fois « tt »)

```
$ grep -E '((.)\2).*\1' /usr/share/dict/french
```

Même question mais sans prendre en compte les « `ss` » et « `nn` » ?

Sous-expressions imbriquées

- On peut imbriquer des sous-expressions
- Et réutiliser les sous-chaînes associées

Question

Quels mots ont 2 fois la même paire de lettres répétées ?
Exemple: « `gouttelette` » (deux fois « tt »)

```
$ grep -E '((.)\2).*\1' /usr/share/dict/french
```

Même question mais sans prendre en compte les « `ss` » et « `nn` » ?

```
$ grep -E '(([^sn])\2).*\1' /usr/share/dict/french
```

Transformations

La capture est utile pour les transformations

- sed
- rechercher/remplacer des éditeurs

Exemple: Inverser les deux premières lettres de chaque ligne

```
$ sed -E 's/^(.)(.)/\2\1/' fruits.txt
abnane
amngue
opmme
```

Pour `sed`, l'**esperluette** « `&` » dénote le motif complet

```
$ echo "8 chiens 15 chats" | sed -E 's/[0-9]+/*&*/g'
*8* chiens *15* chats
```

Question

Extraire la partie centrale des mots qui

- Commencent par « inter »
- Et finissent par « aux ».
- Exemple: « inter**continent**aux » $\longrightarrow$ « continent »

Question

Extraire la partie centrale des mots qui

- Commencent par « inter »
- Et finissent par « aux ».
- Exemple: « inter**continent**aux » $\longrightarrow$ « continent »

```
$ sed -En 's/^inter(.*)aux$/\1/p' /usr/share/dict/french
```

Assertion avant et arrière

PCRE permet de définir des contextes (*lookahead*, *lookbehind*)

- `?=` assertion positive en avant
- `?<=` assertion positive en arrière
- `?!` assertion négative en avant
- `?<!` assertion négative en arrière

Les assertions ne consomment pas les caractères

```
$ grep -Po '(?<=inter).*(?=aux)' /usr/share/dict/french
```

Question

Rechercher tous les mots qui contiennent au moins une fois chaque voyelle `a`, `e`, `i`, `o`, `u`. (le retour!)

Assertion avant et arrière

PCRE permet de définir des contextes (*lookahead*, *lookbehind*)

- `?=` assertion positive en avant
- `?<=` assertion positive en arrière
- `?!` assertion négative en avant
- `?<!` assertion négative en arrière

Les assertions ne consomment pas les caractères

```
$ grep -Po '(?<=inter).*(?=aux)' /usr/share/dict/french
```

Question

Rechercher tous les mots qui contiennent au moins une fois chaque voyelle `a`, `e`, `i`, `o`, `u`. (le retour!)

```
$ grep -P '(?=.*a)(?=.*e)(?=.*i)(?=.*o)(?=.*u).*'\
> /usr/share/dict/french
```

Aller plus loin

On a déjà vu beaucoup de puissance

- POSIX BRE et ERE
- Les extensions GNU
- Un peu de PCRE

PCRE (et d'autres normes)

- Réinitialisation « `\K` »
- Options de chaînes « `(?i)` »
- Sous-chaines nommées
- Référence arrière récursives
- Contrôle de la marche-arrière (*backtracking*)
- Structures conditionnelles

→ voir la doc PCRE

XKCD

Source: https://xkcd.com/208/ (2007)

Limites des expressions régulières

Ne permet pas de tout faire

Exemple classique: les constructions récursives ne fonctionnent pas

- HTML, XML
- JSON
- Expression arithmétiques avec des parenthèses

C'est une limite théorique

Rapidement illisible

- `^\(?\d{3}\)?[ -]*\d{3}[ -]*\d{4}$`

Limites des expressions régulières

Ne permet pas de tout faire

Exemple classique: les constructions récursives ne fonctionnent pas

- HTML, XML
- JSON
- Expression arithmétiques avec des parenthèses

C'est une limite théorique

Rapidement illisible

- `^\(?\d{3}\)?[ -]*\d{3}[ -]*\d{4}$` $\longrightarrow$ numéros de téléphone
- `^M{,4}(CM|CD|D?C{,3})(XC|XL|L?X{,3})(IX|IV|V?I{,3})$`

Limites des expressions régulières

Ne permet pas de tout faire

Exemple classique: les constructions récursives ne fonctionnent pas

- HTML, XML
- JSON
- Expression arithmétiques avec des parenthèses

C'est une limite théorique

Rapidement illisible

- `^\(?\d{3}\)?[ -]*\d{3}[ -]*\d{4}$` $\longrightarrow$ numéros de téléphone
- `^M{,4}(CM|CD|D?C{,3})(XC|XL|L?X{,3})(IX|IV|V?I{,3})$`
  $\longrightarrow$ nombres romains valides
- `^@%*&+#$`

Limites des expressions régulières

Ne permet pas de tout faire

Exemple classique: les constructions récursives ne fonctionnent pas

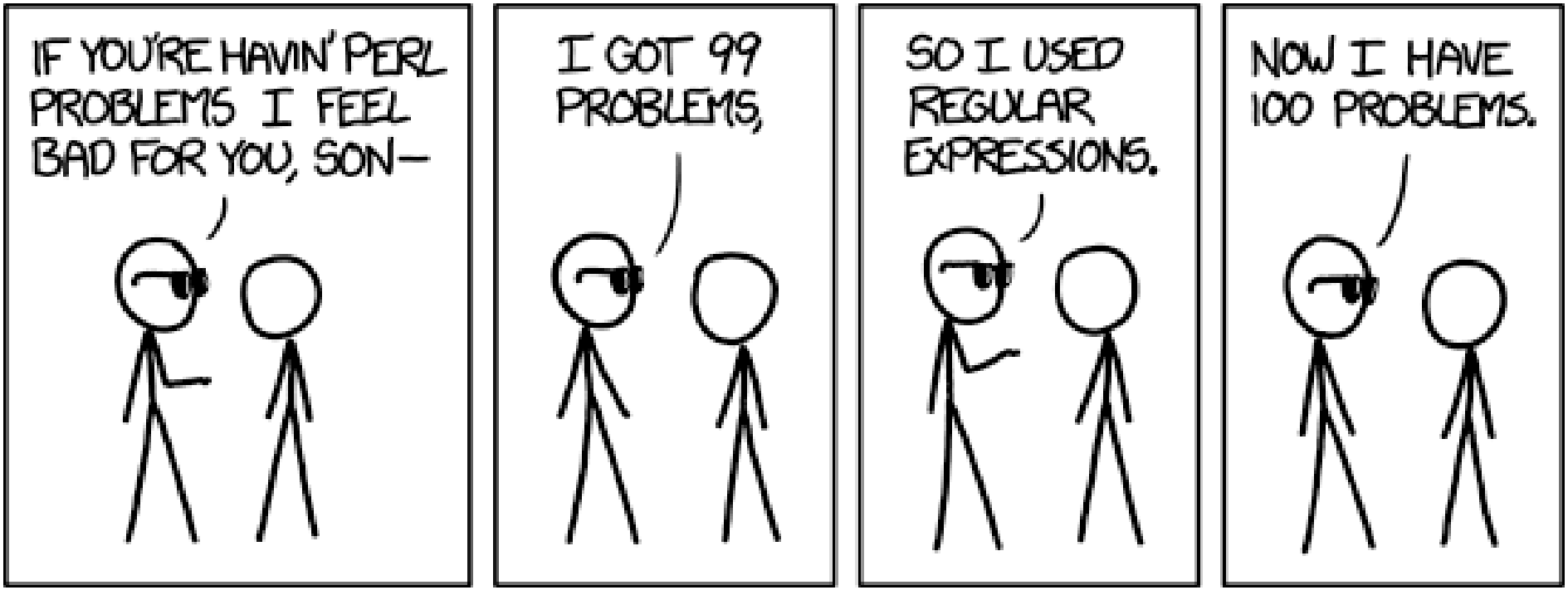
- HTML, XML
- JSON
- Expression arithmétiques avec des parenthèses

C'est une limite théorique

Rapidement illisible

- `^\(?\d{3}\)?[ -]*\d{3}[ -]*\d{4}$` $\longrightarrow$ numéros de téléphone
- `^M{,4}(CM|CD|D?C{,3})(XC|XL|L?X{,3})(IX|IV|V?I{,3})$`
  $\longrightarrow$ nombres romains valides
- `^@%*&+#$` $\longrightarrow$ sans doute un juron

XKCD



Source: https://xkcd.com/1171/ (2013)

Théorie des langages

Expressions régulières

Formellement, une expression régulière est construite à partir

- D'un **alphabet** et de la chaîne **vide**
- Des opérateurs de **concaténation**, d'**alternance** et de l'**étoile de Kleene**

Complexité des langages

Plusieurs types de langages formels: Hiérarchie de Chomsky

- Langages réguliers (*regular*, type 3)
- Langages algébriques (*context free*, type 2)
- Langages contextuels (*context sensitive*, type 1)
- Langages généraux (type 0)

Comment ça marche ?

- Expressions régulières utilisent des automates à états finis
  $\rightarrow$ Très efficaces
- Grammaires algébriques utilisent des automates à pile
  $\rightarrow$ Très efficaces aussi
- Les captures et extensions PCRE utilisent des algorithmes récursifs et du *backtrack*
  $\rightarrow$ Potentiellement très inefficaces

En savoir plus

- INF5000: Théorie et construction des compilateurs
- INF5030: Théorie des automates
- INF600E: Création de langages informatiques