INF 421 — 08 Luc Maranget

Expressions régulières (rationnelles)

Luc.Maranget@inria.fr
http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421/

Plan

- Qu'est-ce qu'un mot ?
- Comment décrire des ensembles simples de mots ?
- Programmation de tout ça.

À quoi ça sert ? À jouer aux mots-croisés (entre autres) ! Mais aussi :

- À définir des notations, telles que l'écriture des **int** dans un source Java.
- À effectuer des recherches et des modifications dans du texte.
- À nettoyer votre répertoire.
- Donne un bon exemple de la distinction syntaxe/sémantique.

Jeux avec les mots

Soient A et B deux humains, A joue aux mots croisés, B l'aide :

- A pose une question : il ne manque pas d'air, en sept lettres la seconde est `x`, l'avant-dernière est `n` ? `oxygène`
- Une autre : livre sacré en cinq lettres, commence par `b` ou `c` ? `bible` ou `coran`.

Dans les questions de A, il y a une partie typiquement humaine, et une autre plus simple.

Si B est un ordinateur, on peut donner des *motifs*...

- `.x...n.` (« `.` » est une lettre quelconque).
- `(b|c)....` (« `|` » est l'alternative).

Si on dispose d'un fichiers de mots, on peut extraire les mots qui conviennent par `egrep`, par exemple `egrep -x '.x...n.'` *`file`*

La répétition

Pour se donner plus de liberté, on introduit `*`

- `.*` : tous les mots (répétition de `.`).
  - Les mots qui contiennent (au moins) deux `k` : `.*k.*k.*`
- Si p motif, alors p`*` : répétition du motif.
  - Les mots dont les lettres sont prises dans `abcd` : `(a|b|c|d)*`.
  - Lettres prises dans `abcd` et `efgh` alternativement :`((a|b|c|d)(e|f|g|h))*`.

Essayez pour voir :

```
# egrep -x '.*k.*k.*' /usr/share/dict/french
bookmaker
bookmakers
  ...
```

Les mots

- Soit Σ un *alphabet* (ensemble de caractères).
- Les *mots* (ensemble Σ^*) sont les suites finies de caractères.
- Un *langage* est un sous-ensemble de Σ^*.
  - Langage des mots français.
    - $\Sigma = \{\texttt{a}, \texttt{b}, \ldots\}$ (alphabet usuel, avec accents).
    - Défini par, le dictionnaire (de l'Académie, si on y tient).
  - Langage des écritures en base 10.
    - $\Sigma = \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$ (chiffres).
    - Défini par une périphrase genre « le chiffre `0`, ou une suite non-vide de chiffres qui ne commence pas par `0` ».

Opérations sur les mots

Un mot est par exemple $m = a_0 a_1 \cdots a_{n-1}$ (n longueur de m).

- Récupérer le caractère d'indice k : a_k.
- Concaténer deux mots : les mettre l'un derrière l'autre.

$$m \cdot m' = a_0 a_1 \cdots a_{n-1} a'_0 a'_1 \cdots a'_{n'-1}$$

  - Opération associative.
  - Mot vide élément neutre à gauche et à droite ($\epsilon \cdot m = m \cdot \epsilon = m$).
  - Dans $t = m \cdot m'$, m est un *préfixe* de t, m' est un *suffixe* de t.
- Notation abrégée : $t = mm'$.
- Notation des sous-mots :

$$m[i \cdots j[= a_i a_{i+1} \cdots a_{j-1}$$

Les mots sont des chaînes

La classe `String`[a], du package (importé par défaut) `java.lang`.

- Mot vide : `""`.
- Pour récupérer le caractère d'indice k.
  public char `charAt(int index)`
- Concaténer deux chaines :

```
String t = m0 + m1 ;
```

  Ou méthode `concat`.

```
String t = m0.concat(m1) ;
```

- Décomposer en préfixe et suffixe, avec la méthode `substring`.

```
String pref = m.substring(0, k) // m[0···k[
String suff = m.substring(k)    // m[k···n[
```

[a] `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html`

Opérations sur les langages

- Union ensembliste.
- Concaténation

$$L_1 \cdot L_2 = \{m_1 \cdot m_2 \mid m_1 \in L_1 \wedge m_2 \in L_2\}$$

- Itération.

$$L^0 = \{\,\epsilon\,\} \qquad L^{n+1} = L^n \cdot L \qquad L^\star = \bigcup_{i \in \mathbb{N}} L^i$$

Autrement dit :

$$m \in L^\star \iff m = m_1 \cdot m_2 \cdots m_n, \quad \text{avec } m_i \in L$$

Les expressions régulières

Une représentation p d'un langage L.

- Le mot vide ϵ représente $\{\,\epsilon\,\}$.
- Le caractère c représente $\{\,c\,\}$.
- L'alternative $p_1 \mid p_2$ represente $L_1 \cup L_2$.
- La concaténation $p_1 \cdot p_2$ représente $L_1 \cdot L_2$.
- La répétition $p*$ représente $L^\star$.

Pour l'informaticien, la définition mérite une décomposition en *syntaxe* et *sémantique*.

Syntaxe des expressions régulières

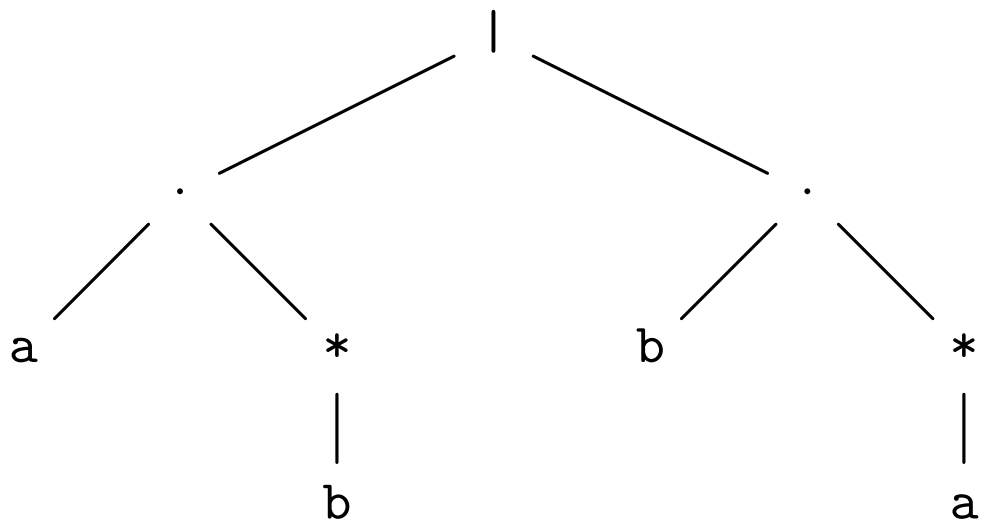
Un motif p est défini ainsi.

- Le mot vide ϵ est un motif.
- Un caractère c est un motif.
- Si p_1 et p_2 sont des motifs, alors...
  - L'alternative $p_1 \mid p_2$ est un motif.
  - La concaténation $p_1 \cdot p_2$ est un motif.
- Si p est un motif, alors...
  - La répétition $p*$ est un motif.

Important C'est une définition de structure d'arbre, sans plus de précisions.

$$P = \{\,\epsilon\,\} \uplus \Sigma \uplus (P \times \{\,\mid, \cdot\,\} \times P) \uplus (P \times \{\,*\,\})$$

Exemple de motif



En pratique on écrit (syntaxe concrète)

(a·(b*))|(b·(a*)) ou plus simplement ab*|ba*

Priorités des opérateurs

C'est comme pour les expressions arithmétiques. Par ordre croissant de priorité.

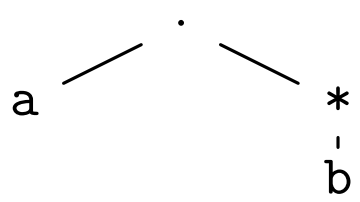
- L'alternative « `|` » (comme l'addition).
- La concaténation « · » (comme la multiplication, peut être omis).
- La répétition « `*` » (postfixe, comme la mise à la puissance).

Donc

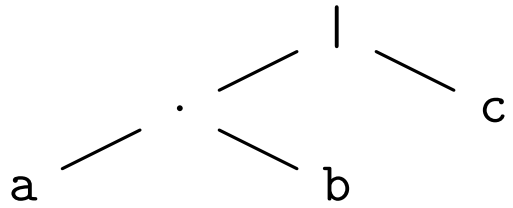
- `ab*` se comprend comme `a(b*)`.
- `ab|c` se comprend comme `(ab)|c`.
- `ab*|c` se comprend comme `(a(b*))|c`.

Encore plus précis

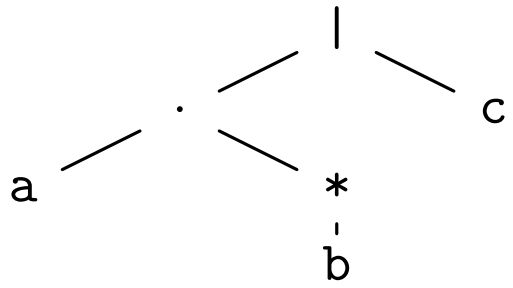
- `ab*` se comprend comme `a(b*)`.



- `ab|c` se comprend comme `(ab)|c`.

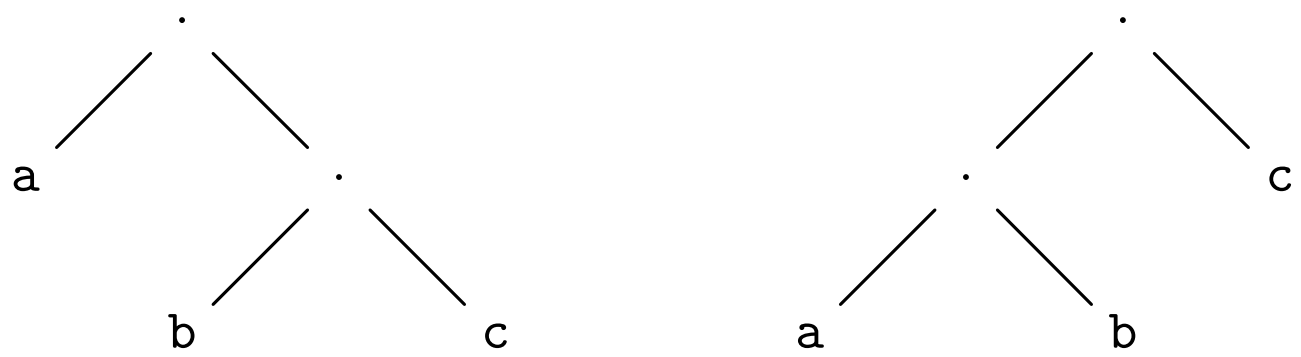


- `ab*|c` se comprend comme `(a(b*))|c`.



Les priorités ne résolvent pas tout

Comment comprendre `abc`, comme `a(bc)` ou `(ab)c` ?



Cela n'a au final pas d'importance, en raison de l'associativité de la concaténation.

Mais il y a bien deux arbres possibles.

Un dernier détail : on peut représenter le motif vide par `()`.

Sémantique des expressions régulières

La fonction $[\![\cdot]\!]$, définie des motifs dans les ensembles de mots, associe une valeur à chaque motif.

$$\begin{aligned}
[\![\epsilon]\!] &= \{\epsilon\} \\
[\![c]\!] &= \{c\} \\
[\![p_1 \mid p_2]\!] &= [\![p_1]\!] \cup [\![p_2]\!] \\
[\![p_1 \cdot p_2]\!] &= [\![p_1]\!] \cdot [\![p_2]\!] \\
[\![p*]\!] &= [\![p]\!]^*
\end{aligned}$$

Inversement,

> Un langage L est dit *régulier* quand il existe un motif p tel que $[\![p]\!] = L$.

Exemples de langages réguliers

- Les mots français.

  Grosse alternative des mots du dictionnaire (de l'Académie)

- Tout langage fini est régulier (même principe).

- Écriture décimale des entiers.

  `0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*`

- Mots constitués de `0` et de `1` alternés
  - ▷ Commence par `0`, finit par `1` : `(01)*` (ou vide).
  - ▷ Commence par `0`, finit par `0` : `(01)*0`.
  - ▷ etc.

  `(01)*|(01)*0|(10)*|(10)*1`

Ou mieux `(|1)(01)*(|0)`

Mots constitués de `0` et `1` alternés.

- Commence par `0`, finit par `1` : `(01)*` (ou vide).
- Commence par `0`, finit par `0` : `(01)*0`.
- Commence par `1`, finit par `1` : `1(01)*`.
- Commence par `1`, finit par `0` : `1(01)*0`.

Soit `(01)*|(01)*0|1(01)*|1(01)*0`.

$$(L \cdot L_1) \cup (L \cdot L_2) = L \cdot (L_1 \cup L_2) \qquad L \cdot \{\epsilon\} = L$$

Soit `((01)*|(|0))|(1(01)*(|0))`.

$$(L_1 \cdot L) \cup (L_2 \cdot L) = (L_1 \cup L_2) \cdot L \qquad \{\epsilon\} \cdot L = L$$

Soit `(|1)(01)*(|0)`.

Filtrage

Si $m \in [\![p]\!]$, on dit que p *filtre* m, noté $p \preceq m$.

Définition formelle par axiomes et règles d'inférences.

$$\textsc{Empty}\quad \epsilon \preceq \epsilon \qquad\qquad \textsc{Char}\quad c \preceq c$$

$$\textsc{Seq}\quad \frac{p_1 \preceq m_1 \qquad p_2 \preceq m_2}{p_1 p_2 \preceq m_1 m_2}$$

$$\textsc{OrLeft}\quad \frac{p_1 \preceq m}{p_1 \mid p_2 \preceq m} \qquad\qquad \textsc{OrRight}\quad \frac{p_2 \preceq m}{p_1 \mid p_2 \preceq m}$$

$$\textsc{StarEmpty}\quad p\!* \preceq \epsilon$$

$$\textsc{StarSeq}\quad \frac{p \preceq m_1 \qquad p\!* \preceq m_2}{p\!* \preceq m_1 m_2}$$

Arbre de dérivation

Un tel arbre donne une preuve, par exemple de $\texttt{ab*|ba*} \preceq \texttt{baa}$.

$$\dfrac{\dfrac{\texttt{b} \preceq \texttt{b} \qquad \dfrac{\dfrac{\texttt{a} \preceq \texttt{a} \qquad \texttt{a*} \preceq \texttt{a}}{\texttt{a*} \prec \texttt{aa}}}{\texttt{ba*} \preceq \texttt{baa}}}{\texttt{ab*|ba*} \preceq \texttt{baa}} \qquad \dfrac{\texttt{a} \preceq \texttt{a} \qquad \texttt{a*} \preceq \epsilon}{\texttt{a*} \preceq \texttt{a}}}$$

C'est un premier emploi de la définition formelle du filtrage.

Autres emplois (des règles d'inférence).

- Implémentation (naïve) du filtrage.
- Preuves par inductions sous l'hypothèse $p \preceq m$.

Notations supplémentaires

Exprimables avec les constructions de base, elle n'étendent pas la classe des langages réguliers.

- Le *joker* « `.` », $[\![.]\!] = \Sigma$.

  $\Sigma = \{a_0, \ldots a_{n-1}\}$ fini, exprimable comme a_0 `|` $\cdots$ `|` a_{n-1}.

- L'*option* p`?`, définie comme p `|` `()`.

- La *répétiton au moins une fois* p`+`, définie comme pp`*`.

- Les ensembles de caractères, écrits entre crochets `[`$\cdots$`]`.

  Par exemple `[aeiou]`, ou `[0-9]` (suppose un ordre sur les caractères).

- Les complémentaires, écrits `[^`$\cdots$`]`.

  Par exemple `[^a-zA-Z0-9]` (caractères non alphanumériques).

Notations supplémentaires

Il y a aussi les *citations*, introduites par « \ ».

- Caractères spéciaux : `\n` (et `\r`), `\t` etc.
- Citer les caractères actifs des motifs : `\*` est un motif qui filtre le caractère étoile, `\[` le crochet ouvrant, `\\` la barre oblique inverse etc.

Les entiers décimaux.

`0|[1-9][0-9]*`, ou plus tolérant `[0-9]+`

Les entiers de Java

- *A decimal numeral is either the single ASCII character 0, representing the integer zero, or consists of an ASCII digit from 1 to 9, optionally followed by one or more ASCII digits from 0 to 9.*
- *A hexadecimal numeral consists of the leading ASCII characters 0x or 0X followed by one or more ASCII hexadecimal digits. Hexadecimal digits with values 10 through 15 are represented by the ASCII letters a through f or A through F, respectively.*
- *An octal numeral consists of an ASCII digit 0 followed by one or more of the ASCII digits 0 through 7.*

```
(0|[1-9][0-9]*)|0[xX][0-9a-fA-F]+|0[0-7]+
```

Conclusion, pour Java, `0` est décimal, `0x0` est hexadécimal, et `00` est octal.

Les commentaires de Java

Première sorte de commentaire

- Commence par `//`.
- Se poursuit jusqu'à la fin de la ligne.

```
//[^\n]*\n
```

Seconde sorte de commentaire.

- Commence par `*/`.
- Se termine par `*/`.
- Ne peut pas être imbriqué.

```
/\*([^*]|\*+[^*/])*\*+/
```

Le shell et les expressions régulières

Ces expressions régulières s'appliquent aux noms de fichier. La syntaxe est spéciale.

- L'expression « `*` » désigne « `.*` » ($\Sigma^\star$).
- Le joker est « `?` ».
- L'alternative est notée `{...,...}`.

Exemples

- Effacer les fichier objets Java `rm *.class`.
- Lister les fichiers dont le nom fait de un à trois caractères, `ls {?,??,???}`.
- Compter les lignes d'un ensemble de sources Java `wc -l *.java`.

Recherche de lignes dans un fichier

La commande `egrep`

`% egrep` *motif* nom_1 nom_2 `...`

Affiche les *lignes* de nom_1 nom_2 ... dont un *sous-mot* est filtré par *motif*.

- Les mots français qui contiennent (au moins) six « `i` ».

  ```
  % egrep 'i.*i.*i.*i.*i.*i' /usr/share/dict/french
  indivisibilité
  ```

- (Re-)trouver la classe Java qui contient la méthode `main`.

  ```
  % egrep main *.java
  ```

Changements automatisés

La commande `sed -e 's/`*p*`/`*m*`/g'` permet de remplacer les mots qui filtrés par p par m.

Exemple : le dictionnaire sans diacritiques français :

```
# sed -e 's/à\|â/a/g' -e 's/ë\|é\|ê\|è/e/g' \
    -e 's/ï\|î/i/g' -e 's/ô/o/g' -e 's/ü\|û\|ù/u/g' \
    -e 's/ç/c/g' /usr/share/dict/french
```

Syntaxe bizarre de l'alternative : « `\|` ».

Implémentation des expressions régulières

- Transformer la représentation textuelle (syntaxe concrète) en arbre (syntaxe abstraite) ▷ 431.
- Implémentation des arbres de syntaxe abstraite.
- Implémentation du filtrage — deux méthodes.
  - À partir des règles.
  - Par la dérivation de motif.

Classe des arbres de syntaxe abstraite

C'est du déjà-vu (amphi 05)

```
class Re {
  private final static int EMPTY=0, CHAR=1, WILD=2,
    OR=3, SEQ=4, STAR=5 ;

  private int tag ;
  private char asChar ;
  private Re p1, p2 ;

  private Re(int tag) { this.tag = tag ; }
    ...
}
```

- Le champ `tag` indique la nature de chaque cellule d'arbre et commande ceux des champs qui sont utiles (par ex. `p1` quand `tag` vaut `STAR`).
- Tout est **private**.

Fabriquer les nœuds de l'arbre

On reprend le modèle « *factory* » (des méthodes statiques de la classe `Re` qui appellent le constructeur).

```
private Re empty() { return  Re (EMPTY) ; }

static empty = empty() ; // Pour éviter quelques allocations

static Re charPat(char c) { // Impossible à nommer « char »
  Re r = new Re(CHAR) ; r.asChar = c ;
  return r ;
}

static Re star(Re p) {
  Re r = new Re(STAR) ; r.p1 = p ;
  return p ;
}
...
```

Rien n'empêche...

D'ajouter d'autres méthodes statiques « de construction ».

```
static Re plus(Re p) {
  return seq(p, star(p)) ;
}

static Re opt(Re p) {
  return or(empty(), p) ;
}
```

Le motif `i.*....*i`

```
static Re buildAtLeast(int k, char c) {
  if (k <= 0) {
    return Re.empty() ;
  } else if (k == 1) {
    return Re.charPat(c) ;
  } else {
    return
      Re.seq
       (Re.charPat(c),
        Re.seq(Re.star(Re.wild()), buildAtLeast(k-1, c)))
  }
}
```

Appeler par ex. `atLeast(6, 'i')`

Écrire `EGrep` nous même, `main`

Que du classique (normalement).

```
class EGrep {
  public static void main(String [] arg) {
    try {
      BufferedReader in =
        new BufferedReader (new FileReader (arg[1])) ;
      grep(arg[0], in) ;
      in.close() ;
    } catch (IOException e) {
      System.err.println("Malaise: " + e.getMessage()) ;
      System.exit(2) ;
    }
  }
  ⋮
}
```

Méthode grep

```
// Affiche les lignes de in dont un sous-mot est filtré par p
static void grep(String p, BufferedReader in)
  throws IOException {
    Re pat = Re.parse(p) ; // ▷ 431
    String line = in.readLine() ;
    while (line != null) {
      if (submatches(pat, line)) {
        System.out.println(line) ;
      }
      line = in.readLine() ;
    }
  }
```

Enfin le vrai travail ?

Supposons donnée, dans la classe `Re` (pourquoi ? tout est privé dans `Re`), une méthode

```
static boolean matches(Re pat, String text, int i, int j)
```

Qui nous dit si `pat` filtre le sous-mot `text`$[i \ldots j[$, ou pas.

Alors, il suffit d'essayer tous les sous-mots de `text`.

```
static boolean submatches(Re pat, String text) {
  for (int i = 0 ; i <= text.length() ; i++)
    for (int j = i ; j <= text.length() ; j++)
      if (Re.matches(pat, text, i, j)) return true
  return false ;
}
```

Filtrage, cas faciles

```
// Test  de pat ≼ text[i...j[
static boolean matches(String text, Re pat, int i, int j) {
  switch (pat.tag) {
    case EMPTY:
      return i == j ;
    case CHAR:
      return i+1 == j && text.charAt(i) == pat.asChar ;
    case WILD:
      return i+1 == j ;
    ⋮
  }
  throw new Error ("Arbre Re incorrect") ;
}
```

L'alternative

Essayer les deux règles possibles.

$$\textsc{OrLeft}\ \frac{p_1 \preceq m}{p_1 \mid p_2 \preceq m} \qquad \textsc{OrRight}\ \frac{p_2 \preceq m}{p_1 \mid p_2 \preceq m}$$

```
case OR:
  return
    matches(text, pat.p1, i, j) ||
    matches(text, pat.p2, i, j) ;
```

La séquence

Essayer la règle

$$\textsc{Seq}\ \frac{p_1 \preceq m_1 \qquad p_2 \preceq m_2}{p_1 p_2 \preceq m_1 m_2}$$

Pour toutes les divisions $m_1 = t[i \cdots k[, m_2 = t[k \cdots j[$.

```
case SEQ:
  for (int k = i ; k <= j ; k++) {
    if (matches(text, pat.p1, i, k) &&
        matches(text, pat.p2, k, j))
      return true ;
  }
  return false ;
```

La répétition

Deux règles,

$$\textsc{StarEmpty}\quad p\!*\preceq\epsilon \qquad\qquad \textsc{StarSeq}\quad \frac{p\preceq m_1 \qquad p\!*\preceq m_2}{p\!*\preceq m_1 m_2}$$

Il faut appliquer la seconde « de toutes les façons possibles ».

```
case STAR:
  if (i == j) {
    return true ;
  } else {
    for (int k = i+1 ; k <= j ; k++) {
      if (matches(text, pat.p1, i, k) &&
          matches(text, pat, k, j))
        return true ;
    }
    return false ;
  }
```

Un détail qui a son importance

- On a écrit **for** `(`**int** `k = i+1 ;...`
- et non pas **for** `(`**int** `k = i ;...`

Pourquoi ?

- La méthode `matches` risquerait de ne pas terminer, en raison de `...&& matches(text, k, j)`.
- Certes, mais sommes nous alors *complets* ?
- Oui, ne pas se servir de la règle :

$$\frac{q \preceq \epsilon \qquad q* \preceq m}{q* \preceq m}$$

ne risque pas de nous empêcher de prouver $q* \preceq m$.

Expressions régulières en Java

Classes `Pattern`[a] et `Matcher`[b] du package `java.util.regexp`.

Architecture en deux classes :

- `Pattern` est une abstraction (on ne sait pas comment c'est implémenté) des motifs. Les deux méthodes-clés sont

  public static `Pattern compile(String regex)`

  Fabrique un objet `Pattern` à partir de la représentation textuelle d'un motif.

  public `Matcher matcher(CharSequence input)`

  Fabrique un « filtreur » qui applique le motif **this** à l'entrée `input` (qui peut être un `String`).

[a] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`
[b] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Matcher.html`

Les filtreurs

Un objet `Matcher` cache un motif (**this** lors de l'appel à `matcher`) et une entrée (l'argument `input` de l'appel à `matcher`).

- Pour savoir si `input` est filtré par le motif.

  public boolean `matches()`

- Pour savoir si un sous-mot de `input` est filtré par le motif.

  public boolean `find()`

Par ex. savoir si `sub` est présent deux fois dans `text`.

```
static boolean twice(String sub, String text) {
  Pattern pat = Pattern.compile(sub + ".*" + sub) ;
  Matcher m = pat.matcher(text) ;
  return m.find() ;
}
```

Justification de la classe `Matcher`

Les objets `Matcher` cachent un état qui évolue lors du filtrage.

- Une position courante dans le texte filtré. Après un appel réussi à `find` cette position est celle qui suit le sous-mot filtré.

```
static boolean twice(String sub, String text) {
  Pattern pat = Pattern.compile(sub) ;
  Matcher m = pat.matcher(text) ;
  if (!m.find()) return false
  return m.find() ;
}
```

- On peut retrouver le sous-mot filtré avec la méthode `group`.

```
static void allInts(String text) { // Tous les entiers de text
  Matcher m = Pattern.compile("[0-9]+").matcher(text) ;
  while (m.find()) {
    System.out.println(m.group()) ;
  }}
```

Un problème sémantique

Sommes nous bien sûr que `allInts` affiche tous les entiers de `text` ?

Autrement dit, quel sous-mot de `text` est filtré par *`"[0-9]+"`* ?

$\underline{12}_1\ \underline{34}_2$ $\qquad$ $\underline{1}_1\underline{2}_2\ \underline{3}_3\underline{4}_4$ $\qquad$ $12\ \underline{34}_1$

Deux règles possibles :

- `find` choisit un sous-mot *le plus à gauche* (élimine le troisième cas).
- Parmi les sous-mots les plus à gauche, `find` choisit *le plus long* (élimine le second cas).

Attention, la bibliothèque Java a une autre seconde règle. Elle spécifie que `+` est *avide*, ici cela revient au même.

Retour sur notre `submatches`

Le voici, qui renvoie le sous-mot filtré.

```
static String submatches(Re pat, String text) {
  for (int i = 0 ; i <= text.length() ; i++) // Plus à gauche, ok.
    for (int j = i ; j <= text.length() ; j++) // Plus court
      if (Re.matches(pat, text, i, j))
        return text.substring(i,j) ;
  return null ;
}
```

Code pour le plus à gauche–le plus long.

```
static String submatches(Re pat, String text) {
  for (int i = 0 ; i <= text.length() ; i++) // Plus à gauche, ok.
    for (int j = text.length() ; j >= i ; j--) // Plus long
      if (Re.matches(pat, text, i, j))
         return text.substring(i,j) ;
  return null ;
}
```

Dérivation des motifs

Soit L langage et c caractère, on définit le langage dérivé $c^{-1} \cdot L$.

$$c^{-1} \cdot L = \{m \mid c \cdot m \in L\}$$

Exemples

- $\mathtt{a}^{-1} \cdot \{\mathtt{a}\} = \{\,\epsilon\,\}$.
- $\mathtt{a}^{-1} \cdot \{\mathtt{b}\} = \emptyset$.
- $\mathtt{a}^{-1} \cdot \{\mathtt{ab}\,\} = \{\,\mathtt{b}\,\}$.
- $\mathtt{a}^{-1} \cdot \{\mathtt{ab}, \mathtt{ba}\,\} = \{\,\mathtt{b}\,\}$.
- $\mathtt{a}^{-1} \cdot [\![\mathtt{(ab)*}]\!] =$
  - $[\![\mathtt{(ab)*}]\!] = \{\,\epsilon, \mathtt{ab}, \mathtt{abab}, \mathtt{ababab}, \ldots\,\}$
  - Donc la dérivation est : $\{\,\mathtt{b}, \mathtt{bab}, \mathtt{babab}, \ldots\,\}$
  - Soit $\mathtt{a}^{-1} \cdot [\![\mathtt{(ab)*}]\!] = [\![\mathtt{b(ab)*}]\!]$

Théorème (carrément)

Si L est régulier, alors $c^{-1} \cdot L$ est régulier.

C'est à dire il faut trouver un motif $c^{-1} \cdot p$ tel que

$$[\![c^{-1} \cdot p]\!] = c^{-1} \cdot [\![p]\!]$$

Si le thérorème est vrai (et si nous savons décider $p \preceq \epsilon$) il permet de décider le filtrage de m par p facilement.

1. Si $m = \epsilon$
   (a) Si $p \preceq \epsilon$, alors renvoyer *vrai*.
   (b) Sinon, renvoyer, *faux*.
2. Si $m = cm'$, poser $p = c^{-1} \cdot p$ et $m = m'$, aller en 1.

Dérivation, cas faciles

Commençons par étendre les motifs, soit $\emptyset$ le motif tel que $[\![\emptyset]\!] = \emptyset$.

$$
\begin{array}{rcll}
c^{-1} \cdot \emptyset & = & \emptyset & \\
c^{-1} \cdot \epsilon & = & \emptyset & \\
c^{-1} \cdot c & = & \epsilon & \\
c^{-1} \cdot c' & = & \emptyset & \text{pour } c \neq c' \\
c^{-1} \cdot (p_1 \mid p_2) & = & (c^{-1} \cdot p_1) \mid (c^{-1} \cdot p_2) &
\end{array}
$$

Décider $p \preceq \epsilon$

Définir un prédicat $\mathcal{N}$ tel que $p \preceq \epsilon \iff \mathcal{N}(P)$.

$$\mathcal{N}(\emptyset) = faux \qquad \mathcal{N}(\epsilon) = vrai \qquad \mathcal{N}(c) = faux$$

$$\mathcal{N}(p_1 \mid p_2) = \mathcal{N}(p_1) \vee \mathcal{N}(P_2) \qquad \mathcal{N}(p_1 \cdot p_2) = \mathcal{N}(p_1) \wedge \mathcal{N}(P_2)$$

$$\mathcal{N}(p*) = vrai$$

Bout de preuve.

- Si $p = p_1 \cdot p_2$, il y a une et une seule façon d'appliquer la règle SEQ.

$$\frac{p_1 \preceq \epsilon \qquad p_2 \preceq \epsilon}{p_1 \cdot p_2 \preceq \epsilon}$$

D'où, $p_1 \cdot p_2 \preceq \epsilon \iff p_1 \preceq \epsilon \wedge p_2 \preceq \epsilon$. Conclure par induction.

Dérivation, $p*$

$$c^{-1} \cdot p* = (c^{-1} \cdot p) \cdot p*$$

Preuve Par définition :

$$\frac{\textsc{StarSeq} \quad p \preceq m_1 \qquad p* \preceq m_2}{p* \preceq m_1 m_2}$$

Et donc :

$$p* \preceq c \cdot m \iff \bigwedge \begin{cases} m = m_1 \cdot m_2 \\ p \preceq c \cdot m_1 \\ p* \preceq m_2 \end{cases}$$

Conclure par induction ($p \preceq c \cdot m_1 \iff c^{-1} \cdot p \preceq m_1$).

Dérivation, $p_1 \cdot p_2$

La règle :

$$\textsc{Seq}\ \frac{p_1 \preceq m_1 \qquad p_2 \preceq m_2}{p_1 p_2 \preceq m_1 m_2}$$

Analysons $p_1 \cdot p_2 \preceq c \cdot m$. Deux sous-cas:

- Si p_1 ne peut pas filtrer ϵ ($\mathcal{N}(p_1) = \mathit{faux}$).

$$p_1 \cdot p_2 \preceq c \cdot m \iff \bigwedge \begin{cases} p_1 \preceq c \cdot m_1 \\ p_2 \preceq m_2 \end{cases} \iff \bigwedge \begin{cases} c^{-1} \cdot p_1 \preceq m_1 \\ p_2 \preceq m_2 \end{cases}$$

- Sinon, il y a un cas supplémentaire.

$$\bigwedge \begin{cases} p_1 \preceq \epsilon \\ p_2 \preceq c \cdot m \end{cases} \iff \bigwedge \begin{cases} p_1 \preceq \epsilon \\ c^{-1} \cdot p_2 \preceq m \end{cases}$$

Dérivation, récapitulatif

$$
\begin{array}{rcll}
c^{-1}\cdot\emptyset & = & \emptyset & \\
c^{-1}\cdot\epsilon & = & \emptyset & \\
c^{-1}\cdot c & = & \epsilon & \\
c^{-1}\cdot c' & = & \emptyset & \text{pour } c \neq c' \\
c^{-1}\cdot(p_1 \mid p_2) & = & (c^{-1}\cdot p_1) \mid (c^{-1}\cdot p_2) & \\
c^{-1}\cdot(p_1 \cdot p_2) & = & (c^{-1}\cdot p_1)\cdot p_2 & \text{si } p_1 \not\preceq \epsilon \\
c^{-1}\cdot(p_1 \cdot p_2) & = & ((c^{-1}\cdot p_1)\cdot p_2) \mid (c^{-1}\cdot p_2) & \text{si } p_1 \preceq \epsilon \\
c^{-1}\cdot p* & = & (c^{-1}\cdot p)\cdot p* &
\end{array}
$$