TD2 - Langages réguliers

Exercice 1

Donner une expression régulière correspondant à chacun des langages suivants, décrits en français.

Question 1. Les mots sur $\Sigma = \{a, b\}$ contenant exactement un a.

Question 2. Les mots sur $\Sigma = \{a, b\}$ contenant toujours un b directement après un a.

Question 3. Les mots sur $\Sigma = \{a, b, c\}$ ne contenant pas de a après le premier b.

Question 4. Les mots sur $\Sigma = \{a, b, c\}$ où chaque paire de a est séparée par exactement 3 caractères.

Exercice 2

Dans un langage de programmation, on restreint le nom des variables en exigeant qu'elles ne soient formées que des lettres $\{a, b\}$, du symbole _, des chiffres $\{0, 1\}$ et qu'elles commencent par une lettre et ne se terminent pas par _. Trouver une expression régulière sur l'alphabet $\Sigma = \{a, b, 0, 1, \_\}$.

Exercice 3

Soit r une expression régulière. Montrer que $(r^*)^*$, $(r|\varepsilon)^*$ et r^* sont équivalentes.

Exercice 4

Sur l'alphabet $\Sigma = \{a, b\}$, montrer que le langage L des mots contenant exactement deux fois la lettre b est régulier.

Exercice 5

Soit $\Sigma = \{a, b\}$ un alphabet et a et b deux de ses lettres. Donner une description des langages dénotés par les expressions régulières suivantes :

$$(\varepsilon|\Sigma)(\varepsilon|\Sigma) \qquad (\Sigma^2)^* \qquad (b|ab)^*(a|\varepsilon) \qquad (ab^*a|b)^*$$

Exercice 6

Question 1. Montrer que l'intersection des langages dénotés respectivement par $(b^*a^2b^*)^*$ et $(a^*b^2a^*)^*$ est régulier.

Question 2. Montrer que le complémentaire du langage dénoté par $(a|b)^*b$ est régulier.

Exercice 7

Le langage $L = \{a^n b^m \mid (n + m) \equiv 0 \mod 2\}$ est-il régulier ?

Exercice 8

Montrer que tout langage fini est régulier.

Exercice 9

Cet exercice a pour but de coder directement un évaluateur d'expressions régulières. On se base pour celà sur le type OCaml défini ci-dessous. On donne également la fonction `has_epsilon : re -> bool` qui renvoie `true` si et seulement si l'expression régulière passée en argument accepte le mot vide.

```
type re =
  | Empty
  | Epsilon
  | Char of char
  | Alt of re * re
  | Concat of re * re
  | Star of re

let rec has_epsilon (r : re) : bool =
  match r with
  | Empty -> false
  | Epsilon -> true
```

```
  | Char c -> false
  | Alt (re1, re2) -> has_epsilon re1 || has_epsilon re2
  | Concat (re1, re2) -> has_epsilon re1 && has_epsilon re2
  | Star _ -> true
```

On considère que l'alphabet Σ est l'ensemble des caractères représentables par le type `char` d'OCaml. Par exemple, l'expression régulière ab^*a est représentée par la valeur :

```
let re1 = Concat(Char 'a', Concat(Star (Char 'b'), Char  'a'))
```

L'algorithme que nous proposons d'implémenter est basé sur la notion de *dérivée d'un langage pour un mot*, concept introduit par l'informaticien polonais canadien Janusz Antoni Brzozowski (1935–2019). Soit $L \subseteq \Sigma^*$ un langage et $v \in \Sigma^*$ un mot. La dérivée $v^{-1}L$ de l'ensemble L pour le mot v est l'ensemble des mots w tels que $vw \in L$. Autrement dit, étant donné un préfixe v, la dérivée de L est l'ensemble des façons de compléter v pour obtenir un mot de L.

Nous illustrons informellement l'algorithme sur un exemple. Supposons que l'on veuille savoir si le mot `aba` est reconnu par l'expressions ab^*a.

- on prend la première lettre du mot, `a`. L'ensemble des façons de continuer ce mot pour être dans le langage est donné par l'expression b^*a ;
- on passe à la seconde lettre du mot. L'ensemble des façons de continuer le mot `b` pour être dans le langage b^*a est $b^*a|a$. En effet, pour tout mot de ce langage qui commence par un `b`, la fin du mot peut être soit une séquence de `b` suivie d'un `a`, soit un `a`.
- on passe à la troisième lettre du mot. L'ensemble des façons de continuer le mot `a` pour être dans $b^*a|a$ est ε.

L'expression dérivée finale contient le mot vide, donc le mot est accepté par l'expression régulière initiale. Si l'expression dérivée finale ne contient pas le mot vide, cela signifie qu'en lisant lettre à lettre le mot v et en avançant dans l'expression régulière, on arrive à un point où il faut forcément lire une lettre supplémentaire, donc le mot v n'appartient pas au langage de l'expression.

Question 1. Écrire une fonction `derivative : re -> char -> re` telle que `derivative re c` est l'expression dérivée de `re` pour le caractère `c`. Le cas le plus subtil est celui de la concaténation. En effet, Considérons l'expression r_1r_2 et le caractère c. Si r_1 contient ε, alors les dérivés de r_1r_2 sont ceux de r_1 concaténé à r_2 ou directement ceux de r_2. Pour l'étoile de Kleene, on pourra remarquer que les dérivés de r^* pour un caractère c sont les dérivés de r pour c concaténés à r lui-même. On utilisera la fonction `has_epsilon` pour tester qu'une expression reconnaît le mot vide.

Question 2. En déduire une fonction `bmatch : re -> string -> bool` qui renvoie `true` si et seulement si la chaîne passée en argument appartient au langage de l'expression régulière donnée.

Question 3. Quelle est la complexité de cette fonction ? On pourra par exemple considérer l'expression $(a|a^*)^*b$ et la chaîne $\underbrace{a \dots a}_{n \text{ fois}}b$

Question 4. Proposer une façon d'éviter le cas précédent.