Exercices de type B

Exercice 5 *Langages locaux (type B)*

Consignes : Cet énoncé est accompagné d'un code compagnon en OCaml localite.ml *fournissant le type décrit par l'énoncé et quelques fonctions auxiliaires : il est à compléter en y implémentant les fonctions demandées. On privilégiera un style de programmation fonctionnel.*

On considère un alphabet Σ. Si L est un langage sur Σ, on note :

- $P(L) = \{a \in \Sigma \mid a\Sigma^\star \cap L \neq \emptyset\}$ l'ensemble des premières lettres des mots de L.
- $D(L) = \{a \in \Sigma \mid \Sigma^\star a \cap L \neq \emptyset\}$ l'ensemble des dernières lettres des mots de L.
- $F(L) = \{m \in \Sigma^2 \mid \Sigma^\star m \Sigma^\star \cap L \neq \emptyset\}$ l'ensemble des facteurs de longueur 2 des mots de L.
- $N(L) = \Sigma^2 \setminus F(L)$ l'ensemble des mots de taille 2 qui ne sont pas facteurs de mots de L.

On rappelle qu'un langage L est dit *local* si et seulement si l'égalité de langages suivantes est vérifiée :

$$L \setminus \{\varepsilon\} = (P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L) \Sigma^\star)$$

1. Calculer les ensembles $P(L), D(L), F(L)$ et $N(L)$ dans le cas où L est le langage dénoté par l'expression régulière $a^\star(ab)^\star + aa$ sur l'alphabet $\{a, b\}$. Ce langage est-il local ? On vérifiera la cohérence entre les réponses à cette question et celles obtenues via les fonctions demandées dans la suite de l'énoncé.

On cherche dans la suite de l'exercice à concevoir un algorithme répondant à la spécification suivante :

Entrée : Une expression régulière e sur un alphabet Σ ne faisant pas intervenir le symbole $\emptyset$.
Sortie : Vrai si le langage dénoté par e est local, faux sinon.

Par défaut, dans la suite de l'énoncé, "expression régulière" signifie "expression régulière ne faisant pas intervenir le symbole $\emptyset$". Les expressions régulières seront manipulées en OCaml via le type somme suivant :

```
type regexp =
  | Epsilon
  | Letter of string (*La chaîne en question sera toujours de longueur 1*)
  | Union of regexp * regexp
  | Concat of regexp * regexp
  | Star of regexp
```

On propose tout d'abord de calculer les ensembles $P(L)$, $D(L)$ et $F(L)$ à partir d'une expression régulière dénotant L. Ces ensembles seront représentés en OCaml par des listes de chaînes de caractères qui vérifieront les propriétés suivantes :

- Elles sont triées dans l'ordre croissant selon l'ordre lexicographique.
- Elles sont sans doublons.

L'énoncé fournit une fonction `union` permettant de calculer l'union sans doublons de deux listes triées. La définition inductive d'une expression régulière invite à calculer inductivement les ensembles $P(L)$, $D(L)$ et $F(L)$. C'est ce que propose la fonction `compute_P` fournie par l'énoncé.

2. En supposant que la fonction `contains_epsilon : regexp -> bool` renvoie `true` si et seulement si le langage dénoté par l'expression régulière en entrée contient le mot ε, justifier brièvement la correction de `compute_P`.
3. Implémenter la fonction `contains_epsilon`.

4. Sur le modèle de `compute_P`, implémenter une fonction `compute_D : regexp -> string list` permettant le calcul de l'ensemble $D(L)$ étant donnée une expression régulière dénotant le langage L.
5. Expliquer en langage naturel comment calculer récursivement l'ensemble $F(L)$ étant donnée une expression régulière e dénotant le langage L. Si $e = e_1e_2$ on pourra exprimer $F(L)$ en fonction notamment de $P(L_2)$ et $D(L_1)$ où L_1 (resp. L_2) est le langage dénoté par e_1 (resp. e_2).
6. Écrire une fonction `prod : string list -> string list -> string list` calculant le produit cartésien des deux listes en entrée, qu'on pourra supposer triées dans l'ordre lexicographique croissant et sans doublons, puis qui pour chaque couple de chaînes dans la liste obtenue les concatène. Par exemple :

```
prod ["a";"c";"e"] ["b";"c"] = ["ab";"ac";"cb";"cc";"eb";"ec"]
```

7. En déduire une fonction `compute_F : regexp -> string list` déterminant l'ensemble $F(L)$ étant donnée une expression régulière dénontant le langage L.

Dans les questions qui suivent, on ne demande PAS d'implémenter les algorithmes décrits.

8. Décrire en pseudo-code ou en langage naturel un algorithme permettant de calculer un automate reconnaissant $(P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L)\Sigma^\star)$ étant donnée une expression régulière dénotant L.
9. Décrire un algorithme permettant de détecter si le langage dénoté par une expression régulière est local ou non.
10. Pourquoi est-il légitime de ne considérer que les expressions régulières ne faisant pas intervenir $\emptyset$? Comment modifier l'algorithme obtenu dans le cas où cette contrainte n'est plus vérifiée ?

Proposition de corrigé

1. On obtient $P(L) = \{a\}$, $D(L) = \{a, b\}$, $F(L) = \{aa, ab, ba\}$ et donc $N(L) = \{bb\}$. Le langage L n'est pas local : s'il l'était, il devrait contenir le mot aba ce qui n'est pas.
2. La seule difficulté est de justifier la disjonction de cas pour les concaténations. Si $e = e_1e_2$ et $\varepsilon \notin L(e_1)$, une première lettre d'un mot de e est nécessairement une première lettre d'un mot de e_1 (la réciproque étant évidente). Sinon, un mot de $L(e)$ peut aussi commencer de la même façon qu'un mot de $L(e_2)$.
3. On filtre sans grande difficulté.

```
let rec contains_epsilon (e:regexp) :bool = match e with
  |Epsilon -> true
  |Letter _ -> false
  |Union (e1, e2) -> (contains_epsilon e1) || (contains_epsilon e2)
  |Concat (e1, e2) -> (contains_epsilon e1) && (contains_epsilon e2)
  |Star _ -> true
```

4. Comme pour `compute_P`, la seule difficulté est le cas d'une concaténation.

```
let rec compute_D (e:regexp) :string list  = match e with
  |Epsilon -> []
  |Letter a -> [a]
  |Union (e1, e2) -> union (compute_D e1) (compute_D e2)
  |Concat (e1, e2) when contains_epsilon e2 ->
     union (compute_D e1) (compute_D e2)
  |Star e1 |Concat (_, e1) -> (compute_D e1)
```