TP-Colle-2 (MPI*) :

Environnement

Nous travaillerons en OCaml mode interprété pour ce TP-colle. Nous vous laissons le choix de l'environnement, selon vos habitudes. Vous pouvez utiliser :

- L'interpréteur en ligne BetterOCaml : https://BetterOCaml.ml/
- Emacs, muni du mode Tuareg.

Consignes et mise en place

- Veuillez trouver dans le répertoire partagé le fichier `regexp.ml` (aussi disponible sur la page Cahier-de-Prépa : `https://cahier-de-prepa.fr/info-kleber/`). Si vous utilisez un interpréteur en ligne, importez-y ce fichier, ou copiez-son contenu.
- Vous répondrez aux questions de programmation en écrivant le code dans votre outil.
- Vous répondrez aux questions de cours à l'aide de la fiche-réponse à la fin du sujet.
- Vous répondrez aux questions à développer à l'oral lors de votre passage au tableau (pour les concernés). Utilisez un brouillon pour vous préparer des notes.

I Préliminaires

1. *(Question de cours)* Montrez qu'un langage est régulier si et seulement si il est engendré par une expression régulière.
2. Déclarez un type `'a exp_reg`, basé sur la définition inductive d'une expression régulière. (Vous pouvez choisir d'inclure ε comme une expression de base.)
3. Implémentez une fonction `accepte_vide : 'a exp_reg -> bool` qui teste si une expression régulière accepte le mot vide ou non.

II Langages résiduels

Si Σ est un alphabet, u un mot sur Σ et L un langage sur Σ, on définit le *langage résiduel de L par u* : $u^{-1}L = \{v \mid uv \in L\}$.

1. *(Question à développer à l'oral)* Si $e = (aab)^*$, donnez une expression régulière qui engendre le langage $(aaba)^{-1}L(e)$ et une qui engendre le langage $(aab)^{-1}L(e)$

   On appelle *dérivée d'une expression e par rapport à une lettre a* toute expression régulière qui engendre le langage $a^{-1}L(e)$.
2. Implémentez une fonction `derivee : 'a exp_reg -> 'a -> 'a exp_reg` telle que `derivee e a` génère une expression régulière `e'` dérivée de `e` pour la lettre `a`.
3. *(Question à développer à l'oral)* Exprimez la complexité de votre fonction par rapport à la taille de l'expression.
4. *(Question à développer à l'oral)* Montrez que pour tous mots $f, g \in \Sigma^* \backslash \{\varepsilon\}$ et tout langage L sur Σ, $(fg)^{-1}L = g^{-1}(f^{-1}L)$
5. Proposez une méthode pour vérifier si un mot u appartient au langage engendré par une expression régulière e. Implémentez-la dans une fonction `accepte : 'a exp_reg -> 'a mot -> bool`
6. *(Question à développer à l'oral)* Exprimez la complexité de votre fonction par rapport à la taille de l'expression.
7. L'expression générée par `derivee` de manière naïve peut être simplifiée. Imaginez des règles de simplification d'une expression, et implémentez les dans une fonction `simplifie`.

III Préfixes, suffixes et facteurs dans une expression

Un langage L sur Σ est local s'il existe 3 ensembles $P \subset \Sigma, D \subset \Sigma$ et $N \subset \Sigma^2$ tel que :

$$L\backslash\{\varepsilon\} = (P\Sigma^* \cap \Sigma^* D)\backslash(\Sigma^* N \Sigma^*)$$

La connaissance de P, D et N suffit donc à caractériser L. Alternativement, on peut le caractériser à partir de P, D et $F = \Sigma^*\backslash N$.

1. Dans la construction de l'automate de Glushkov, on linéarise des expressions régulières en ajoutant des indices distincts à chaque occurrence de lettre. Ecrire une fonction `linearise` (dont vous spécifierez la signature) qui transforme une expression régulière en expression régulière linéaire.

   On sait qu'une expression linéaire engendre un langage local. Ceci intervient dans la construction de l'automate de Glushkov. On se propose d'effectuer la construction des ensembles P (premières lettres des mots du langage), D (dernieres lettres des mots du langage) et F (facteurs de longueur 2 intervenant dans les mots du langage) et d'évaluer la complexité de cette construction.
2. *(Question à développer à l'oral)* Etant donné une expression régulière linéaire, définissez un ensemble de règles de récurrence pour le calcul de P, D et F.
3. Implémentez deux fonctions `premieres : 'a exp_reg -> 'a list`, `dernieres : 'a exp_reg -> 'a list` construisant les ensembles P, D pour une expression régulière linéaire.
4. Implémentez une fonction `facteurs : 'a exp_reg -> ('a mot) list` construisant l'ensemble F pour une expression régulière linéaire.
5. *(Question à développer à l'oral)* Estimez la complexité des fonctions précédentes.
6. *(Question à développer à l'oral)* En proposant une structure de données adaptée estimez la complexité de la construction de l'automate de Glushkov.

   On généralise la notion de langage local. Un langage est dit *k-testable* s'il existe 3 ensembles $P \subset \Sigma^{k-1}, D \subset \Sigma^{k-1}$ et $N \subset \Sigma^k$ tel que :
   $$L\backslash\{\varepsilon\} = (P\Sigma^* \cap \Sigma^* D)\backslash(\Sigma^* N \Sigma^*)$$
   On peut poser de même $F = \Sigma^k\backslash N$ et se servir de P, D, F pour caractériser un tel langage. Un langage local est donc un langage 2-testable.
7. On note $P(e,k)$ (resp. $D(e,k), F(e,k)$) l'ensemble des préfixes (resp. suffixes, facteurs) de longueur k des mots engendrés par e. Proposer des règles de récurrence pour le calcul de ces ensembles.
8. Implémentez des fonctions réalisant les calculs précédents.

IV Algorithme de Conway, matrices régulières

L'algorithme de Conway est un algorithme qui permet de calculer une expression rationnelle reconnaissant le même langage qu'un automate fini. Cet algorithme construit des matrices dont les coefficients sont des expressions rationnelles. On s'intéresse ici aux manipulations sur ces matrices, que nous appellerons *matrices régulières*. (Nous vous encourageons à tester vos fonctions essentiellement avec des matrices 2×2 si le temps vous manque.)

1. Définissez un type `'a mat_reg` représentant des matrices d'expressions régulières.
2. Définissez une fonction `somme : 'a mat_reg -> 'a mat_reg -> 'a mat_reg` qui, pour deux matrices $A = (e_{ij})_{0\leq i<n;0\leq j<p}$ et $B = (f_{ij})_{0\leq i<n;0\leq j<p}$ de même taille, construit la matrice $(e_{ij} \mid f_{ij})_{0\leq i<n;0\leq j<p}$
3. Pour une taille n donnée, peut on trouver une matrice B telle que, avec les notations précédentes, pour tout i et tout j, $e_{ij} \mid f_{ij}$ engendre le même langage que e_{ij} ? Définissez une fonction `nulle : int -> int -> 'a mat_reg` construisant cette matrice, dont la taille est donnée en argument.
4. Définissez une fonction `produit : 'a mat_reg -> 'a mat_reg -> 'a mat_reg` qui, pour deux matrices $A = (e_{ij})_{0\leq i<n;0\leq j<p}$ et $B = (f_{ij})_{0\leq i<p;0\leq j<q}$ de même taille, construit la matrice du produit AB avec la règle suivante : le produit matriciel est calculé de la même manière que pour des matrices prises sur un corps K, où l'union joue le rôle de la somme et la concaténation celui du produit sur K.

On cherche maintenant à définir l'étoilée d'une matrice carrée régulière. Si $M = (e)$ est une matrice de taille 1, alors $M^* = (e^*)$. Sinon, M s'écrit par blocs $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$, et la matrice M^* est définie par :

$$\begin{pmatrix} (A \mid BD^*C)^* & A^*B(D \mid CA^*B)^* \\ D^*C(A \mid BD^*C)^* & (D \mid CA^*B)^* \end{pmatrix}$$

On admet que le choix de la taille des blocs dans M n'a aucune incidence sur les langages engendrés par les expressions de la matrice régulière finale.

5. *(Question à développer à l'oral)* On appelle $C(n)$ la complexité du calcul de l'étoile pour une matrice de taille n. Si on applique une méthode naïve où A est un bloc de taille 1, qu'obtient-on pour $C(n)$?
6. *(Question à développer à l'oral)* Proposez une méthode plus efficace qu'à la question précédente pour le calcul de l'étoile. Quelle nouvelle complexité obtient-on ?
7. *(Question à développer à l'oral)* En admettant qu'on dispose d'une fonction `decoupe : 'a exp_reg -> int -> int` qui renvoie un quadruplet de matrice correspondance à la décomposition par bloc, et d'une fonction `recolle` qui reconstitue la matrice à partir de ses 4 blocs, écrire sous-forme de pseudo-code lam éthode précédente.
8. Implémentez les deux fonctions précédentes, ainsi que le calcul de l'étoile.

L'algorithme de Conway construit une "matrice de transition" d'un automate : si les états sont numérotés de 0 à $n-1$, le coefficient m_{ij} de la matrice est l'union de toutes les lettres qui étiquettent la transition de i vers j, ou $\emptyset$ s'il n'en existe pas. Le calcul de l'expression régulière correspondant à l'automate utilise ensuite l'étoile de cette matrice.

Annexes

Vous trouverez ici des rappels concernant certains traits du langage OCaml

Module Array

- `Array.init n f` crée le tableau `[| f(0) ; f(1) ; ...; f(n-1)|]`
- `Array.make n k` crée un tableau de taille `n` dont tous les éléments valent `k`
- `Array.make_matrix n p k` crée un tableau à deux dimensions, avec `n` lignes et `p` colonnes, dont tous les éléments initialisés à `k`
- `Array.length tab` renvoie la longueur du tableau `tab`
- `tab.(i)` récupère le `i`-ème élement du tableau `t`
- `Array.mem x tab` renvoie un booléen spécifiant si `x` apparaît dans `tab`
- `Array.map f tab` renvoie le tableau `[| f(a.(0)) ; f(a.(1)) ; ...; f(a.(n-1))|]`
- `Array.iter f tab` exécute dans cet ordre `f(a.(0)), f(a.(1)), ..., f(a.(n-1))`

Fiche réponse

NOM :
Prénom :
Répondez ici aux questions de cours, en rédigeant vos réponses.