Projet *À remettre au plus tard le 31 mars*

Le projet doit être écrit en utilisant la portion de Ocaml étudiée en cours. Vous pouvez, sans les redéfinir, utiliser les fonctions prédéfinies sur les listes `mem`, `map`, `fold_left` et `right`, `find`, `find_all`, `exists` et `for_all`.

Il sera apprécié que vous utilisiez des fonctions d'ordre supérieur dès que cela est pertinent.

Le projet est individuel. Toute ressemblance suspecte entre projets sera sanctionnée.

Le but du projet est de définir une expression régulière correspondant à un automate fini déterministe (AFD) en utilisant la méthode par élimination d'états. Les deux diapos du cours de théorie des langages et compilation qui concernent cette méthode vous sont redonnées dans le document joint.

Un AFD sera représenté par :

- le nombre d'états (on supposera que les états sont numérotés de 0 à nb_etats-1)

- l'état initial

- la liste des états acceptants

- la fonction de transition donnée sous forme d'une liste de couples **(e, ls)** où **e** est un état (type **int**) et **ls** la liste de ses successeurs (type **('a*int) list**).

Dans l'AFD initial, les transitions porteront sur des caractères (type char).

Cet AFD sera ensuite transformé en un AFD généralisé dans lequel les transitions porteront sur des expressions régulières.

Dans cet AFD généralisé, pour chaque état acceptant, les états seront éliminés un à un (sauf l'état initial et cet état acceptant) ; de cet automate réduit résultera une expression régulière. L'expression régulière finale est alors obtenue en faisant la disjonction des expressions régulières obtenues pour les différents états acceptants.

Une expression régulière sera représentée grâce au type suivant :

```
type expr_reg =    Vide
                 | Epsilon
                 | S of char
                 | Ou of expr_reg * expr_reg
                 | Concat of expr_reg * expr_reg
                 | Etoile of expr_reg
;;
```

Consignes

Vous devez déposer sur Moodle le ou les fichiers contenant votre projet. Il devra contenir une brève présentation de votre travail : ce qui est fait (ou non), ce qui fonctionne correctement (ou non), des choix que vous avez effectués le cas échéant, et tout commentaire pertinent.

Chaque fonction doit être commentée (son rôle, ses arguments, son résultat et toute explication qui peut aider la lecture et la compréhension de votre travail). Les noms des fonctions et autres identificateurs doivent également permettre de faciliter la lecture.

Vous devez bien structurer le problème et écrire des fonctions (clairement documentées) pour chaque sous-tâche que vous identifierez. En particulier :

1. Écrire une fonction qui, étant donné un graphe dont les transitions sont étiquetées par des caractères, retourne le même graphe dont les transitions sont étiquetées par des expressions régulières (on transforme un AFD ordinaire en AFD généralisé).
2. Écrire une fonction qui, étant donné une expression régulière (de type **expr_reg**), retourne la même expression donnée sous forme d'une chaîne de caractères en utilisant les opérateurs **|**

(pour la disjonction), **.** (pour la concaténation) et *. (La fonction prédéfinie `Char.escaped : char->string` convertit un caractère en chaîne.)

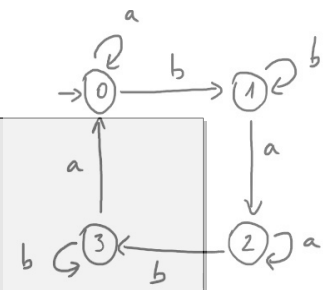
3. Écrire une fonction qui élimine un état d'un graphe généralisé et ajoute les transitions nécessaires entre chacun de ses prédécesseurs et chacun de ses successeurs (autres que lui-même).

   Penser à ne garder qu'une transition entre deux états. (Par exemple, dans l'exemple ci-dessous , s'il y avait initialement une transition de 0 à 2 étiquetée « c » alors, après suppression de l'état 1, la nouvelle transition de 0 à 2 serait étiquetée « c | bb*a ».)

4. Écrire une fonction qui, étant donné un graphe réduit à un état initial et un état final, retourne l'expression régulière correspondante.

5. Et n'oubliez pas d'écrire la fonction demandée qui, étant donné un AFD initial, retourne l'expression régulière correspondante !

   Pensez également à proposer des jeux d'essai pour tester vos fonctions.




```
# let trans = [(0,[('a',0);('b',1)]) ; (1,[('b',1);('a',2)]) ;
             (2,[('a',2);('b',3)]) ; (3,[('a',0);('b',3)])] ;;
 val trans : (int * (char * int) list) list =
  [(0, [('a', 0); ('b', 1)]); (1, [('b', 1); ('a', 2)]);
   (2, [('a', 2); ('b', 3)]); (3, [('a', 0); ('b', 3)])]

# let trans_gene = generaliser_afd trans ;;
 val trans_gene : (int * (expr_reg * int) list) list =
  [(0, [(S 'a', 0); (S 'b', 1)]); (1, [(S 'b', 1); (S 'a', 2)]);
   (2, [(S 'a', 2); (S 'b', 3)]); (3, [(S 'a', 0); (S 'b', 3)])]

# eliminer_un_etat 1 trans_gene ;;
 - : (int * (expr_reg * int) list) list =
[(0, [(Concat (S 'b', Concat (Etoile (S 'b'), S 'a')), 2); (S 'a', 0)]);
 (2, [(S 'a', 2); (S 'b', 3)]); (3, [(S 'a', 0); (S 'b', 3)])]

(* graphe réduit obtenu après élimination de tous les états sauf les états
0 et 3 du graphe à 4 états trans_gene *)
# let trans_reduit = eliminer_tous [0;3] 4 trans_gene ;;
val trans_reduit : (int * (expr_reg * int) list) list =
[(0,[(Concat (S 'b',
       Concat (Etoile (S 'b'),
            Concat (S 'a', Concat (Etoile (S 'a'), S 'b')))),
       3);
     (S 'a', 0)]);
 (3, [(S 'a', 0); (S 'b', 3)])]

(* ER correspondant à un AFD réduit aux états 0 (initial) et 3 (final) *)
# let er03 = expression_reguliere 0 3 trans_reduit ;;
 val er03 : expr_reg =
Concat
 (Concat (Etoile (S 'a'),
   Concat (S 'b',
    Concat (Etoile (S 'b'),
       Concat (S 'a', Concat (Etoile (S 'a'), S 'b'))))),
  Etoile
   (Ou (S 'b',
     Concat (Concat (S 'a', Etoile (S 'a')),
      Concat (S 'b',
       Concat (Etoile (S 'b'),
          Concat (S 'a', Concat (Etoile (S 'a'), S 'b'))))))))

#  expr_reg_to_string er03
- : string = "((a*.(b.(b*.(a.(a*.b))))).(b | ((a.a*).(b.(b*.(a.
(a*.b))))))*)"
```