Implémentation d'un automate fini non déterministe

1 Implémentation

On s'intéresse à un automate fini non déterministe à n états, numérotés de 0 à $n-1$. Un ensemble d'états sera représentée par une liste *triée, sans doublons,* des entiers représentant les états de l'ensemble (par exemple, l'ensemble d'états $\{1,5,2\}$ sera représenté par la liste OCaml `[1; 2; 5]`).

```
type etat = int;;
type liste_etats = etat list;;
```

Les symboles de Σ sont assimilés à des entiers (de 0 à $p-1$).

```
type symbole = int;
```

Un mot est quant à lui représenté par une liste de symboles :

```
type mot = symbole list;
```

On définit un tel AFND en OCaml par le type suivant :

```
type afnd = { initiaux : liste_etats;
              terminaux : liste_etats;
              delta : liste_etats array array };;
```

Les définitions (et ébauches des fonctions) peuvent être récupérés sur `llg.sci-phy.org` .

Dans cette définition, `delta` est un tableau de tableaux à n entiers, de sorte que si e est un état et c un symbole, la fonction d'évolution de l'automate $\delta(e,c)$ appliquée à cet état e pour le symbole c sera `delta.(e).(c)`.

1. Proposer une fonction `union` (`liste_etats -> liste_etats -> liste_etats`), prenant en argument deux liste d'états (respectant les invariants choisis) et renvoyant une liste d'états (toujours respectant les invariants choisis) représenant leur union. On pourra se baser sur cette ébauche :

```
let union lst1 lst2 = match lst2 with
  | [] -> ...
  | t::q when List.mem t lst1 -> ...
  | t::q -> ...
```

Remarque : la signature sera a priori `'a list -> 'a list -> 'a list`.

2. Proposer de même une fonction `intersection` déterminant l'intersection de deux listes d'états.

3. Proposer une fonction `lit_car` de signature `afnd -> liste_etats -> symbole -> liste_etats` prenant en argument un automate $\mathcal{A}$, une liste d'états $\mathcal{L}$ et un symbole c et renvoyant

$$\bigcup_{q\in\mathcal{L}} \delta(q,c)$$

Remarque : la signature sera a priori `afnd -> int list -> int -> int list`.

4. À partir de `lit_car`, créer une fonction `lit_mot` de signature `afnd -> liste_etats -> mot -> liste_etats` prenant en argument un automate $\mathcal{A}$, une liste d'états $\mathcal{L}$ et un mot w et renvoyant

$$\bigcup_{q\in\mathcal{L}} \delta^*(q,w)$$

5. En déduire une fonction `teste` de signature `afnd -> mot -> bool` prenant un automate et un mot, et renvoyant un booléen indiquant si le mot a été accepté ou non par l'automate.

6. Quelle est la complexité de la fonction précédente en fonction de n, p et $|w|$?

2 Construction à partir d'un langage local

On considère un langage local L ne contenant pas de mot vide, défini par les ensembles P, S et F (ensemble des facteurs permis). Par exemple, le langage L_1 des mots non vides sur $\Sigma=\{0,1,2\}$ ne commençant pas par 0, se terminant par 0 et ne contenant pas les facteurs 11, 22 ou 33 est défini par $P=\{1,2\}$, $S=\{0\}$ et $F=\{01,02,10,12,20,22\}$.

On représentera P et S par des `symbole list`, et F par des `(symbole * symbole) list`.

7. Proposer une fonction `construit_local` de signature `int -> symbole list -> symbole list -> (symbole * symbole list) -> afnd` construisant un automate local reconnaissant le langage local défini par p, P, S et F fournis en paramètres, où $p=|\Sigma|$. On associera à un symbole l'état identifié par le même entier (l'état associé au symbole désigné par « 1 » sera numéroté « 1 »). L'état supplémentaire nécessaire sera identifié par p.

8. Vérifier son bon fonctionnement avec L_1 (on construira l'automate associé, et on vérifiera que 10, 210, 231321310 sont reconnus, mais pas ϵ, 12, 012 et 012210 par exemple).

9. On fournit des fonctions permettant, à partir d'une expression régulière *linéaire,* de

construire P, S et F (adaptés d'après le cours). En déduire une fonction `compile_regex` de signature `regexp -> afnd`.

10. Tester la fonction sur l'expression régulière linéaire fournie, qui s'interprète en un langage L_2, en vérifiant que les mots reconnus sont les bons.

3 Déterminisation

On souhaite obtenir un automate fini déterministe équivalent. On représente un automate fini déterministe par

```
type afd = { initial : etat;
             terminaux : liste_etats;
             delta : etat array array };;
```

11. Écrire une fonction `determinise` de signature `afnd -> afd` qui construit l'automate des parties de façon à déterminister l'automate non-déterministe fourni en argument. On remarquera que l'on peut comparer deux listes d'états respectant nos invariants avec `=`! Les listes d'états peuvent par ailleurs être utilisées comme clés de dictionnaire. Aussi utilisera-t-on un dictionnaire pour associer des listes d'états atteints dans l'automate fini déterministe à un entier représentant l'état correspondant dans l'automate fini déterministe (on peut utiliser `Hashtbl.length` pour savoir combien d'états ont déjà été construits).

12. Déterminiser les deux automates fournis.

13. On fourni une fonction `teste_afd`. L'utiliser pour vérifier que les automates déterminisés reconnaissent les mêmes langages que leurs homologues non déterministes.

Pour construire un outil prenant une expression régulière quelconque et fournissant un automate fini déterministe qui en reconnaît l'interprétation, il ne reste donc qu'à ajouter une étape de linéarisation et l'étape permettant de l'inverser avant la déterminisation.

14. Quelle est la complexité de la transformation ?