# TP3 - Déterminisation d'un automate

Dans ce sujet, $\Sigma$ désigne l'alphabet composé de deux lettres $a$ et $b$ : $\Sigma = \{a, b\}$. Le type `alphabet` représente cet alphabet en OCaml à l'aide des deux constructeurs constants `A` et `B` associés aux lettes $a$ et $b$ respetivement. Les deux types suivants définissent le type `nfa` d'un automate fini non déterministe et le type `dfa` d'un automate fini déterministe. Noter la différence dans la définition des fonctions de transitions.

```
(* type alphabet *)
type alphabet = A | B

(* type automate fini non déterministe *)
type nfa = {
  nfa_init : int list;                   (* états initiaux *)
  nfa_finals : int list;                 (* états acceptants *)
  nfa_trans : (int * alphabet * int) list; (* transitions *)
}

(* type automate fini déterministe *)
type dfa = {
  dfa_init : int;                        (* état initial *)
  dfa_finals : int list;                 (* états acceptants *)
  dfa_delta : int * alphabet -> int;     (* transitions *)
}
```

$\mathcal{A}_N = (Q_N, \Sigma, I_N, F_N, \delta_N)$ désigne un automate fini non-déterministe. La construction d'un *automate déterminisé* $\mathcal{A}_D$ repose sur celle de la fonction de transition $\delta_D$ des parties de l'automate $\mathcal{A}_N$. À une partie d'états $P$ de l'automate $\mathcal{A}_N$, représentée par une liste d'entiers, on associe sa *signature* définie par l'entier suivant.

$$s(P) = \sum_{q \in P} 2^{q-1}$$

**Question 1.** On considère l'automate fini non-déterministe suivant.

```
let nfa1 = {
  nd_init = [1; 2];
  nd_finals = [4];
  nd_trans = [ (1, A, 3); (2, B, 2); (2, B, 3); (3, A, 3); (3, A, 4) ]
}
```

□ **1.1.** Représenter cet automate sous forme d'un graphe.
□ **1.2.** Déterminer une expression régulière qui dénote le langage qu'il reconnaît.
□ **1.3.** Le déterminiser et placer sa signature sur chaque partie de l'automate.

**Question 2.**
□ **2.1.** Écrire une fonction `puiss2 : int -> int` telle que `(puiss2 k)` renvoie l'entier $2^k$.
□ **2.2.** Écrire une fonction `signature : int list -> int` qui renvoie la signature d'une partie $P$.
□ **2.3.** Écrire une fonction `list_of_sign : int -> int list` telle que `(list_of_sign s)` renvoie la liste triée d'états correspondant à la signature `s`.

**Question 3.** Un ensemble d'état est représenté par une liste triée sans doublons.

```
type ensemble = int list
```

On donne les fonctions ci-dessous où :

- `ajoute` permet d'ajouter un élément à un ensemble,
- `fusion` permet de fusionner deux ensembles ;
- `(successeurs q c nfa)` renvoie l'ensemble des états $q' \in Q$ tels que $(q, c, q') \in \delta$.

```
(* ajoute un élément à une liste triée sans doublons *)
let rec ajoute i lst =
  match lst with
    | [] -> [i]
    | x :: q when i = x -> lst
    | x :: q when i < x -> i :: lst
    | x :: q -> x :: (ajoute i q)
(* fusionne deux listes triées sans doublons *)
let rec fusion lst1 lst2 =
```

```
  match (lst1, lst2) with
    | [], [] -> []
    | _, [] -> lst1
    | [], _ -> lst2
    | x1 :: q1, x2 :: q2 when x1 = x2 -> x1 :: (fusion q1 q2)
    | x1 :: q1, x2 :: q2 when x1 < x2 -> x1 :: (fusion q1 lst2)
    | x1 :: q1, x2 :: q2 -> x2 :: (fusion lst1 q2)
(* renvoie la liste des états q' d'un nfa tels que (q, l, q') in F *)
let successeurs q c nfa =
  let rec aux lst =
    match lst with
      | [] -> []
      | (i, a, j) :: qlst when i = q && a = c -> ajoute j (aux ql)
      | (i, a, j) :: qlst -> aux ql
  in aux nfa.nd_trans
```

Écrire la fonction de transition des parties `deltapart : ensemble -> alphabet -> nfa -> ensemble` telle que $\delta_D(P, c) = \{q' \in Q \mid \exists q \in P, (q, c, q') \in \delta\}$.

**Question 4.** Un dictionnaire est une structure de données formée d'une liste de couples $l = [(c_1, d_1); \ldots; (c_n, d_n)]$ où $c_i$ est une *clé* (unique) et $d_i$ la *valeur* correspondante. Le module `List` dispose des deux fonctions suivantes :

```
assoc : 'a -> ('a * 'b) list -> 'b
mem_assoc : 'a -> ('a * 'b) list -> bool
```

telles que :

- `List.assoc c dico` renvoie la valeur $d$ correspondant à la clé $c$ ;
- `List.mem_assoc c` teste si la clé $c$ est présente dans le dictionnaire.

Afin de déterminiser un automate $\mathcal{A}_N$, on part de l'ensemble $I$ de ses états initiaux, de signature $q_1$. On calcule $\delta_D(I, a)$ de signature $q_1^a$ et $\delta_D(I, b)$ de signature $q_1^b$. On recommence avec les nouveaux états $q_2 = q_1^a$, $q_3 = q_1^b$ jusqu'à ne plus rencontrer de parties déjà traitées. Ce qui permet la construction du dictionnaire $[(q_1, (q_1^a, q_1^b)); (q_2, (q_2^a, q_2^b)); \ldots]$ dont les clés sont les états de l'automate déterminisé (des signatures d'ensembles), et les données des couples $(q_i^a, q_i^b)$ où $q_i^a = \delta_D(q_i, a)$, $q_i^b = \delta_D(q_i, b)$.
Écrire une fonction `dico_trans : nfa -> (int * (int * int))list` qui renvoie le dictionnaire souhaité. On écrira une fonction auxiliaire `etend q dico` qui prend une signature $q$, le dictionnaire calculé et qui étend ce dictionnaire. Si $q$ est déjà présent, elle renvoie le dictionnaire. Sinon elle ajoute $(q, (q_a, q_b))$ et étend récursivement le dictionnaire avec les états $q_a$ et $q_b$.

**Question 5.** En déduire la fonction `delta_det : dico -> int * alphabet -> int` de sorte que `delta_det dico (q, c)` renvoie l'état $\delta_D(q, c)$.

**Question 6.** Écrire une fonction `finals_det : dico -> nfa -> int list` qui détermine à partir du dictionnaire précédent la liste des états terminaux de l'automate déterminisé. On utilisera la fonction `intersecte : 'a list -> 'a list -> bool` qui teste si deux listes triées ont une intersection non-vide et on supposera que la liste `nfa.nd_init` est triée.

**Question 7.** En déduire la fonction `determinise : nfa -> dfa` qui calcule l'automate déterminisé.

**Question 8.** En déduire également une fonction `reconnait : alphabet list -> dfa -> bool` qui détermine si un mot est reconnu par un automate fini déterministe.