Simulation d'automates

Question 1.

On effectue la division par 2 d'un entier n : le reste donne le premier chiffre (bit de poids faible) dans l'écriture binaire, et on applique récursivement sur le quotient entier $\lfloor n/2 \rfloor$.
L'écriture avec bit de poids fort en premier est simplement un retournement de la liste. On peut également réécrire la fonction en récursive terminale.

```
let rec binaire_faible n = match n with
    | 0 -> []
    | n -> (n mod 2)::(binaire_faible (n / 2)) ;;

let binaire_fort n = List.rev (binaire_faible n) ;;

(* binaire_fort sans List.rev *)
let bin_fort n =
    let rec aux acc n = match n with
        | 0 -> acc
        | n -> aux ((n mod 2):: acc) (n/2)
    in aux [] n;;
```

2. Automates déterministes

Question 2. On va utiliser les fonctions de recherche dans une liste de couples, `List.mem` et `List.assoc`. Rappelons les défintions :
`List.mem x l` teste si l'élément x est dans la liste l : `List.mem : 'a -> 'a list -> bool`
`List.assoc x l` renvoie le second argument v du couple (x, v) présent dans la liste de couples l :
`List.assoc : 'a -> ('a * 'b) list -> 'b`

```
let rec mem x l = match l with
   | []   -> false
   | t::q -> (t=x) || (mem x q) ;;

let rec assoc x l = match l with
   | []                -> failwith "Not_found"
   | (a,v)::q when a=x -> v
   | t :: q            -> assoc x q ;;
```

Pour le parcours de la liste des transitions, de type `(('a * 'b) * 'a) list`, il faut adapter la fonction `mem` afin de tester s'il existe une transition de la forme $((q, a), \_)$:

```
let rec mem_fst x l = match l with
   | [] -> false
   | t :: q -> (fst t = x) || mem_fst x q;;
```

Question 3. L'automate est déterministe, pas nécessairement complet : il n'y a qu'un seul chemin possible pour la lecture d'un mot. Une fonction auxiliaire récursive va prendre progressivement les transitions, en gardant en paramètre le mot restant à lire :

```
let reconnu a mot =
   let rec aux e l = match l with
      | [] -> mem e a.accept              (* le mot est fini, e est un etat acceptant ? *)
      | t :: q when (mem_fst (e,t) a.delta) -> aux (assoc (e,t) a.delta) q
      | _ -> false                        (* aucune transition possible, blocage *)
   in aux a.init mot ;;
```

Question 4. Il s'agit d'implémenter l'automate reconnaissant les mots binaires divisibles par d, en s'inspirant de la construction pour les mots divisibles par 3 (cours), et par 5 (exercice).

Si un mot u est tel que u est congru à k modulo d, et si b est un bit (0 ou 1), alors le mot ub est congru à $2k + b$ modulo d. L'automate possède d états, le seul état initial est 0 (également le seul état final), et les transitions sont décrites par la remarque précédente : $(q, 0, 2 * q \bmod d)$ et $(q, 1, 2 * q + 1 \bmod d)$.
La fonction auxiliaire ici construit les transitions progressivement, à partir de l'état 0 jusqu'à l'état $d - 1$.

```
let genere_fort d =
  let rec aux n = match n with
    | q when q = d -> []
    | q -> [(q, 0), (2*q) mod d; (q, 1), (2*q+1) mod d] @ (aux (q+1))
  in {init = 0; accept = [0]; delta = aux 0} ;;
```

3. Automates non déterministes

Question 5. L'idée est d'échanger le sens de toutes les flèches : $((q_i, a), q_j)$ se transforme en $((q_j, a), q_i)$. Il faut également échanger états finals et état initial, ce qui rend l'automate obtenu non déterministe.

```
let genere_faible d =
  let a = genere_fort d in
  (* aux renvoie les transitions de delta renversées *)
  let rec aux delta = match delta with
    | []             -> []
    | ((q, a), r)::v -> ((r, a), q)::(aux v)
  in {nd_init = a.accept; nd_accept = [a.init]; nd_delta = aux a.delta} ;;
```

Question 6. Ici, il faut parcourir toutes les transitions car plusieurs chemins sont à envisager. Il faut également tester si le mot vide est accepté Enfin, il faut partir de tous les états initiaux : s'il existe un état initial et un chemin acceptant partant de cet état, alors le mot est reconnu. La fonction `List.exists` peut être utile. Rappelons sa définition :
`List.exists pred l` vérifie qu'il existe un élément x dans la liste l tel que pred(x) est vrai.
`List.exists : ('a -> bool) -> 'a list -> bool`

```
let rec exists pred l = match l with
   | [] -> false
   | t ::q -> (pred t) || exists pred q ;;
```

Question 7.

```
1  let reconnu2 a mot =
2    let rec aux e m delta = match (m, delta) with
3      | ([],_) -> mem e a.nd_accept
4      | (_,[]) -> false
5      | (a1::m1, ((q, c), r)::v) when q = e && c = a1 -> aux r m1 a.nd_delta || aux e m v
6      | (m, _::v) -> aux e m v
7  in exists (function e -> aux e mot a.nd_delta) a.nd_init ;;
```

ligne 3 : (* le mot vide est-il reconnu : etat e = etat acceptant ? *)
ligne 4 : (* transitions vides *)
ligne 5 : (* le mot commence par a_1 et la transition est (e, a_1, r), alors soit on prend la transition et on va en r pour lire m_1, soit on ne prend pas la transition et on ne change pas d'état *)
ligne 6 : (* si la transition ne correspond pas à $((e, a_1), \_)$, on parcourt le reste des transitions *)
ligne 7 : on teste chaque etat initial : le mot est reconnu si l'un des etats fournit un chemin acceptant