

TD 12

Automates émondés

Correction

On définit le type automate par :

```
type automate = { initial : int list; finals : int list ; transition : (char*  
int) list array };
```

Exercice 1: Écrire les fonctions utiles au TD :

(-a-) La fonction reconnaît la présence d'une valeur dans une liste :

```
mem : 'a -> 'a list -> bool
```

Correction

Listing 1 – La fonction mem

```
let rec mem x liste = match liste with  
| [] -> false  
| a::reste -> (x=a) || ( mem x reste);;
```

(-b-) La fonction donnant le second membre associé à une valeur dans une liste de couples :

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

Correction

Listing 2 – La fonction assoc

```
let rec assoc x liste = match liste with  
| [] -> raise Not_found  
| (a,b)::reste when a = x -> b  
| (a,b)::reste -> assoc x reste ;;
```

(-c-) la fonction reconnaît la présence du premier terme dans une liste de couples :

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

Correction

Listing 3 – La fonction mem_assoc

```
let rec mem_assoc x l = match l with  
| [] -> false  
| (a,b)::reste when a = x -> true  
| (a,b)::reste -> mem_assoc x reste ;;
```

(-d-) La fonction appliquant à l'ensemble de la liste la fonction f :

```
map : ('a -> 'b) -> 'a list -> 'b list
```

Correction

Listing 4 – La fonction map

```
let rec map f l = match l with
| [] -> []
| a::reste -> (f a)::(map f reste);;
```

(-e-) La fonction renvoyant l'union sans doublon de deux listes :

```
union : 'a list -> 'a list -> 'a list
```

Correction

Listing 5 – La fonction union

```
let rec union l l' = match l' with
| [] -> l
| a::reste when mem a l -> union l reste
| a::reste -> union (a::l) reste;;
```

(-f-) La fonction renvoyant l'intersection de deux listes :

```
intersection : 'a list -> 'a list -> 'a list
```

Correction

Listing 6 – La fonction intersection

```
let rec intersection l l' = match l with
| [] -> []
| a::reste -> if mem a l' then a::(intersection reste l') else
intersection reste l' ;;
```

(-g-) La fonction renvoyant pour deux listes l_1 et l_2 en entrée, les éléments de l_2 qui ne sont pas dans l_1 :

```
difference : 'a list -> 'a list -> 'a list
```

Correction

Listing 7 – La fonction difference

```
let rec difference l1 l2 =
(* ne garde que les elements de l2 qui ne sont pas dans l1 *)
match l2 with
| [] -> []
| a::reste when mem a l1 -> difference l1 reste
| a::reste -> a::(difference l1 reste);;
```

(-h-) La fonction qui transforme une liste de liste en une liste sans doublon :

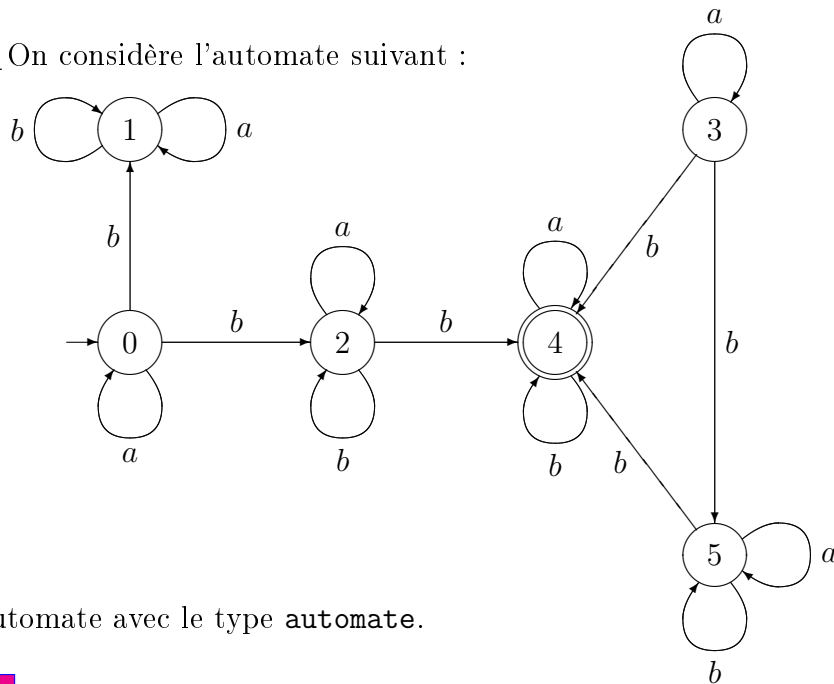
```
list_list : 'a list list -> 'a list
```

Correction

Listing 8 – La fonction list_list

```
let rec list_list l = match l with
| [] -> []
| l1::reste -> union l1 (list_list reste);;
```

Exercice 2 : On considère l'automate suivant :



Définir l'automate avec le type `automate`.

Correction

Listing 9 – L'automate `ex_1`

```

let ex_1 = {initial = [0]; finals = [4];
transition = [|['a', 0; 'b', 1; 'b', 2]; ['a', 1; 'b', 1];
['a', 2; 'b', 2; 'b', 4]; ['a', 3; 'b', 4; 'b', 5]; ['a', 4; 'b', 4];
['a', 5; 'b', 5; 'b', 4]|]};

```

Exercice 3 : Écrire une fonction `trans` qui renvoie pour un r donné, la liste des états s tels que : $\exists a \in A, (r, a, s) \in E$.

`trans : automate -> int -> int list`

Correction

Listing 10 – La fonction `trans`

```

let trans auto p = map snd auto.transition.(p) ;;

```

Exercice 4 : **États accessibles :**

On désire écrire une fonction qui renvoie la liste des états accessibles d'un automate.

Le principe de l'algorithme est simple, on commence par poser $Q_{acc,0} = I$ (les états initiaux sont accessibles par définition).

Si n est un entier et si nous avons déjà calculé $Q_{acc,n}$ à la n ème itération, alors on a :

$$Q_{acc,n+1} = Q_{acc,n} \cup \bigcup_{\substack{q \in Q_{acc,n} \\ a \in A}} \{q' \in Q \mid (q, a, q') \in E\}$$

L'algorithme se termine quand cette suite est constante.

Écrire une fonction `accessible` qui renvoie la liste des états accessibles d'un automate donnée.

`accessible : automate -> int list`

Correction

Listing 11 – La fonction accessible

```

let accessible auto =
  let acces = ref [] in
  let suivants = ref auto.initial in
  while (difference !acces !suivants) <> [] do
    acces := union (!acces) (!suivants);
    suivants := list_list (map (fun p -> trans (auto) p ) (!acces));
  done;
  !acces;;

```

Exercice 5 : États co-accessibles :

- (a) Écrire une fonction `miroir_automate` qui renvoie le miroir d'un automate donné.

`miroir_automate : automate -> automate`

Correction

Listing 12 – La fonction miroir_automate

```

let miroir_automate auto =
  let n = Array.length auto.transition in
  let miroir = Array.make n [] in
  (* la fonction suivante inverse les transitions *)
  let rec aux l i= match l with
    | [] ->()
    | (a,m)::reste -> miroir.(m)<- (a,i)::miroir.(m);aux reste i
  in
  for i = 0 to (n-1) do
    aux (auto.transition.(i)) i
  done;
  {initial = auto.finals; finals=auto.initial; transition=miroir};;

```

- (b) Écrire une fonction `co_accessible` qui renvoie la liste des états co-accessibles d'un automate donné.

`co_accessible : automate -> int list`

Correction

Listing 13 – La fonction co_accessible

```

(* il ne reste plus qu'à appliquer la fonction accessible à l'automate
transpose. *)
let co_accessible auto = accessible ( miroir_automate auto );;

```

Exercice 6 : États utiles :

En déduire une fonction `utile` qui renvoie la liste des états utiles d'un automate donnée :

`utile : automate -> int list`

Correction

Les états utiles sont les états accessibles et co-accessibles

Listing 14 – La fonction utile

```
let utile auto = intersection ( accessible auto ) ( co_accessible auto );;
```

Exercice 7:

- (a) Écrire une fonction `sous_graphe` qui renvoie l'automate privé de l'état i donné (on gardera la même taille pour le vecteur transition dont on remplacera la composante i par la liste vide, et on effacera toutes les transitions avec i) :

`sous_graphe : automate -> int -> automate`

CorrectionListing 15 – La fonction `sous_graphe`

```
let sous_graphe auto i =
(* la fonction qui elimine i d'une liste *)
  let rec aux i l = match l with
    | [] -> []
    | a::reste when a=i -> reste
    | a::reste -> a::(aux i reste)
  in
  let nouveau_init = aux i auto.initial in
  let nouveau_final = aux i auto.finals in
(* la fonction qui elimine tout couple ou i est present *)
  let rec aux2 i l = match l with
    | [] -> []
    | (a,j)::reste when j=i -> aux2 i reste
    | b::reste -> b::(aux2 i reste)
  in
  let n = Array.length auto.transition in
  for h = 0 to n-1 do
    if h=i then auto.transition.(i)<-[] else
      auto.transition.(h)<- aux2 i auto.transition.(h)
  done ;
  {initial = nouveau_init;
   finals = nouveau_final;
   transition = auto.transition};;
```

- (b) En déduire la fonction `emonde` qui renvoie l'automate émondé :

`emonde : automate -> automate`

CorrectionListing 16 – La fonction `emonde`

```
let emonde auto =
  let n = Array.length auto.transition in
  let utiles_etats = utile auto in
  (* la fonction suivante lit tous les etats et ne conserve que les utiles *)
  let rec aux i auto = if i = n then auto else
```

```
    if mem i utiles_etats then aux (i+1) auto else aux (i+1) ( sous_graphe
auto i )
    in
    aux 0 auto;;
```
