Examen (09 janvier 2018)

Durée : 1h45.

Les exercices sont indépendants les uns des autres. Vous pouvez, si nécessaire, admettre le résultat d'une question pour répondre aux suivantes.

Certaines questions sont signalées comme plus difficiles. Il vous est fortement conseillé de les traiter en dernier.

Les documents ne sont pas autorisés.

On pourra considérer, dans tout le sujet, que les fonctions et structures vues en cours sont utilisables. Les profils de certaines fonctions et modules sont rappelés à la fin du sujet.

Exercice 1 Automates finis

Un **vocabulaire** est un ensemble de symboles.

Un **mot** sur un vocabulaire est une suite de symboles de ce vocabulaire.

Un **automate d'états finis** (ou dans la suite simplement automate) $\mathcal{A}$ sur un vocabulaire $\mathtt{V}$ est un quaduplet $(\mathtt{Q}, q_0, \mathcal{F}, T)$ où :

- $\mathtt{Q}$ est un ensemble fini d'**états**,
- $q_0 \in \mathtt{Q}$ est un état dit **état initial**,
- $\mathcal{F} \subset \mathtt{Q}$ est un sous ensemble de $\mathtt{Q}$ dit **ensemble des états acceptants**,
- T est une fonction de $\mathtt{Q} \times \mathtt{V}$ dans $\mathcal{P}(\mathtt{Q})$ l'ensemble des parties de $\mathtt{Q}$.

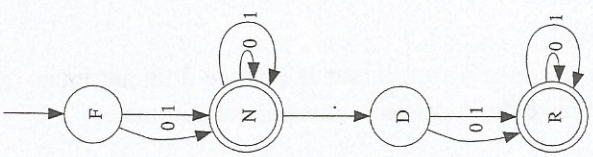
Si la fonction de transition est en fait une fonction de $\mathtt{Q} \times \mathtt{V}$ dans $\mathtt{Q}$, l'automate est dit **déterministe**.

Une représentation graphique des automates est possible (et souhaitable en pratique) en reliant les différents états par des arrêtes étiquetées par les symboles des transitions.

Si nous choisissons comme vocabulaire l'ensemble $\mathtt{V} = \{0, 1, .\}$ nous pouvons définir l'automate (déterministe) $\mathcal{A}$ suivant :

- $\mathtt{Q} = \{F, N, D, R\}$;
- $q_0 = F$;
- $\mathcal{F} = \{N, R\}$;
- $T :=$
$$\begin{cases} (F,0) \to \{N\} \\ (F,1) \to \{N\} \\ \\ (N,0) \to \{N\} \\ (N,1) \to \{N\} \\ (N,.) \to \{D\} \\ \\ (D,0) \to \{R\} \\ (D,1) \to \{R\} \\ \\ (R,0) \to \{R\} \\ (R,1) \to \{R\} \end{cases}$$

dont la représentation graphique [1] est :



1. L'état initial est repéré par la flèche entrante et les états acceptant par leurs doubles cercle.

Si $\mathcal{A} = (\mathrm{V}, \mathrm{Q}, q_0, \mathcal{F}, T)$ est un automate, une **exécution** lisant le mot $u = \alpha_1 \dots \alpha_n$, est une suite d'états $q_0 \to^{\alpha_1} q_1 \dots q_{n_1} \to^{\alpha_n} q_n$ telle que q_0 est l'état initial de $\mathcal{A}$ et $\forall i > 0, q_i \in T(q_{i-1}, \alpha_i)$. Remarquons que tout mot ne conduit pas à une exécution sur $\mathcal{A}$ si la fonction de transition T n'est pas complète (i.e. si il existe un état q et une lettre α tels que $T(q, \alpha) = \emptyset$).

On dit que le mot $u = \alpha_1 \dots \alpha_n$ est **accepté** (ou **reconnu**) par l'automate $\mathcal{A}$ si il existe une exécution de $\mathcal{A}$ lisant u et telle que le dernier état atteint est un état acceptant.

On appelle **langage** reconnu par un automate $\mathcal{A}$ (et l'on note $L(\mathcal{A})$) l'ensemble des mots reconnus par $\mathcal{A}$.

Ainsi, en reprenant automate $\mathcal{A}$ précédent :

- le mot 10.001 est bien reconnu par cet automate puisque l'exécution suivante est valide :

$$F \to^1 R \to^0 N \to^{\cdot} D \to^0 R \to^0 R \to^1 R$$

  et que R est un état final.
- le mot 10. n'est pas reconnu par cet automate puisque la seule exécution lisant le mot 10. est :

$$F \to^1 R \to^0 N \to^{\cdot} D$$

  et que D n'est pas un état final.
- le mot 10..0 n'est pas reconnu puisqu'il n'existe pas d'exécution de $\mathcal{A}$ lisant 10..0

Question 1 : Proposez une définition **efficace** du type `automate` des automates dont le vocabulaire sera le type `char` et les états seront des entiers.

Question 2 : Proposez une fonction `is_det` qui, sur la donnée d'un automate, teste si cet automate est déterministe ou non.

Question 3 : Proposez une définition du type des mots sur un vocabulaire.

Question 4 : Proposez une fonction `build_automaton` qui, sur la donnée d'une liste de triplets q, c, q' (où q et q' sont des états et c est un symbole) représentant chacun une transition valide dans l'automate, d'un état q_0 représentant un état initial et d'une liste $\mathcal{F}$ d'état acceptants, retourne l'automate correspondant.

Question 5 : Proposez une fonction `execute_det` qui, sur la donnée d'un automate `a` supposé déterministe et d'un mot `w`, retourne, si cela est possible, le dernier état atteint par l'exécution de `a` sur `w` et lève une exception que vous définirez sinon.

Question 6 : Proposez une fonction `execute` qui, sur la donnée d'un automate `a` et d'un mot `w`, retourne, l'ensemble des derniers états atteints par l'exécution de `a` sur `w`.

Question 7 : Proposez une fonction `recognize` qui, sur la donnée d'un `a` et d'un mot `w`, teste si le mot `w` est reconnu par `a`

Question 8 : Proposez une fonction `simplify` qui, sur la donnée d'un automate `a`, retourne le sous-automate de `simplify` reconnaissant le même langage que `a` mais dont tous les états sont sur un chemin entre l'état initial et un état final.

Question 9 : Proposez une fonction `is_finite` qui teste si un automate reconnaît un langage fini (i.e. un ensemble fini de mot).

Question 10 :(difficile) Proposez une fonction `enumerate` qui, sur la donnée d'un automate qui sera supposé reconnaître un langage fini, retourne la liste des mots reconnus par cet automate.

```
module type Set =
  sig
    type elt
    (** The type of the set elements. *)

    type t
    (** The type of sets. *)

    val empty: t
    (** The empty set. *)

    val is_empty: t -> bool
    (** Test whether a set is empty or not. *)

    val mem: elt -> t -> bool
    (** [mem x s] tests whether [x] belongs to the set [s]. *)

    val add: elt -> t -> t
    (** [add x s] returns a set containing all elements of [s],
       plus [x]. If [x] was already in [s], [s] is returned unchanged. *)

    val remove: elt -> t -> t
    (** [remove x s] returns a set containing all elements of [s],
       except [x]. If [x] was not in [s], [s] is returned unchanged. *)

    val union: t -> t -> t
    (** Set union. *)

    val inter: t -> t -> t
    (** Set intersection. *)

    val diff: t -> t -> t
    (** Set difference. *)

    val fold: (elt -> 'a -> 'a) -> t -> 'a -> 'a
    (** [fold f s a] computes [(f xN ... (f x2 (f x1 a))...)],
       where [x1 ... xN] are the elements of [s], in increasing order. *)

    val cardinal: t -> int
    (** Return the number of elements of a set. *)

    val compare: t -> t -> int
    (** Total ordering between sets. Can be used as the ordering function
       for doing sets of sets. *)

    val choose: t -> elt
    (** Return one element of the given set, or raise [Not_found] if
      the set is empty. *)
  end

module Make(X:OrderedType) : Set with type elt = X.t
```

```
module type Map =
  sig
    type key
    (** The type of the map keys. *)

    type 'a t
    (** The type of maps from type [key] to type ['a]. *)

    val empty: 'a t
    (** The empty map. *)

    val is_empty: 'a t -> bool
    (** Test whether a map is empty or not. *)

    val mem: key -> 'a t -> bool
    (** [mem x m] returns [true] if [m] contains a binding for [x],
       and [false] otherwise. *)

    val add: key -> 'a -> 'a t -> 'a t
    (** [add x y m] returns a map containing the same bindings as
       [m], plus a binding of [x] to [y]. If [x] was already bound
       in [m] to a value that is physically equal to [y],
       [m] is returned unchanged. Otherwise, the previous binding
       of [x] in [m] disappears. *)

    val remove: key -> 'a t -> 'a t
    (** [remove x m] returns a map containing the same bindings as
       [m], except for [x] which is unbound in the returned map.
       If [x] was not in [m], [m] is returned unchanged. *)

    val fold: (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    (** [fold f m a] computes [(f kN dN ... (f k1 d1 a)...)],
       where [k1 ... kN] are the keys of all bindings in [m]
       (in increasing order), and [d1 ... dN] are the associated data. *)

    val find: key -> 'a t -> 'a
    (** [find x m] returns the current binding of [x] in [m],
       or raises [Not_found] if no such binding exists. *)
  end

module Make(X:OrderedType) : Map with type key = X.t
```

```
module type OrderedType =
  sig
    type t
      (** The type of the set elements. *)

    val compare : t -> t -> int
      (** A total ordering function over the set elements. *)
  end
```

```
val length : 'a list -> int
(** Return the length (number of elements) of the given list. *)

val rev : 'a list -> 'a list
(** List reversal. *)


val map : ('a -> 'b) -> 'a list -> 'b list
(** [List.map f [a1; ...; an]] applies function [f] to [a1, ..., an],
   and builds the list [[f a1; ...; f an]] with the results returned
   by [f]. *)

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
(** [List.fold_left f a [b1; ...; bn]] is
   [f (... (f (f a b1) b2) ...) bn]. *)

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
(** [List.fold_right f [a1; ...; an] b] is
   [f a1 (f a2 (... (f an b) ...))]. Not tail-recursive. *)


val mem : 'a -> 'a list -> bool
(** [mem a l] is true if and only if [a] is equal
   to an element of [l]. *)
```