Compilation d'expressions rationnelles

Nous nous intéressons aujourd'hui au problème de déterminer si une expression rationnelle reconnaît une chaîne de caractères. La traduction naïve de la définition des chaînes reconnues par une expression rationnelle mène à un algorithme par essais et erreurs qui est très inefficace dans certains cas. La manière efficace de déterminer si une expression rationnelle reconnaît une chaîne de caractères est de transformer d'abord l'expression rationnelle en un automate qui reconnaît les mêmes mots, puis d'exécuter l'automate sur ladite chaîne de caractères.

1 Automates non-déterministes et expressions rationnelles

1.1 Expressions rationnelles

On considère un *alphabet* Σ (les éléments de Σ, les *caractères*, sont notés c). Une *expression rationnelle* permet de représenter de manière finie un *langage* sur Σ, c'est-à-dire une partie de l'ensemble des mots sur Σ, usuellement noté Σ^*. Les *expressions rationnelles* r sur Σ sont définies par la grammaire suivante :

$$\begin{array}{rcl} r & ::= & \varepsilon \\ & | & c \\ & | & (r \mid r) \\ & | & (rr) \\ & | & r^* \end{array}$$

On définit le langage dénoté par une expression rationnelle de manière récursive. ε dénote le langage contenant le mot vide ε tandis que c dénote le langage $\{c\}$. Si r_1 et r_2 dénotent respectivement les langages L_1 et L_2 alors $(r_1 \mid r_2)$ dénote l'union des langages : $L_1 \cup L_2$ et $(r_1 r_2)$ dénote la concatenation $L_1 \cdot L_2$. Enfin r_1^* dénote le langage L_1^*.
Dans la suite de cet énoncé, nous supposerons que les éléments de Σ sont représentés en Caml par des *caractères* de type `char`.

▶ **Question 1** *Donner une définition par un type inductif Caml (comme pour les arbres), pour representer les expressions rationnelles (*regular expressions *en anglais).*

Pour pouvoir tester les fonctions construites on va se donner un petit exemple.

▶ **Question 2** *Écrivez en Caml, dans le type définit à la question précedente, l'expression rationnelle correspondant à $a(bc \mid d)^*e$.*

1.2 Construction de Thompson

L'algorithme de transformation d'une expression rationnelle en automate que nous allons employer est connu sous le nom de « construction de Thompson ». Les automates qu'il produit ont la particularité d'avoir un seul état final. De plus aucune transition ne sort de l'état final. En revanche les epsilon transitions sont authorisés. Ainsi, un automate de Thompson pourrat être représenté en Caml par un enregistrement du type suivant :

```
pe thompson =
{
  initial : int ;
  final : int ;
  trans : (int * char * int) list ;
  eps_trans : (int * int) list ;
}
```

La construction de Thompson procède par récurrence sur la structure de l'expression rationnelle. Les questions 3 à 7 introduisent quelques fonctions auxiliaires permettant d'effectuer les opérations de base sur les automates, que vous pourrez ensuite utiliser pour réaliser la construction de Thompson à la question 8.
Au fur et a mesure de la construction on va avoir besoin de nouveau états, comme ils sont représenter de façon unique par un entier, il est neccessaire de connaître ceux qui sont déjà utilisé. Pour cela on peut soit déclarer une reference qui contiendra le nombre d'états, soit passer un argument supplémentaire à chaque fonction et qui contient cette valeur. C'est ce qu'il est suggéré de faire dans la suite

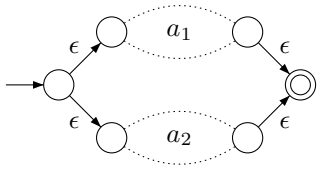
▶ **Question 3** *Écrivez une fonction* `th_epsilon` *telle que* `th_epsilon i` *retourne un automate de Thompson reconnaissant le langage vide.*

```
val th_epsilon: int -> thompson * int
```

▶ **Question 4** *Écrivez une fonction* `th_char` *telle que* `th_char c` *retourne un automate de Thompson reconnaissant le langage $\{c\}$.*

```
val th_char: int -> char -> thompson * int
```

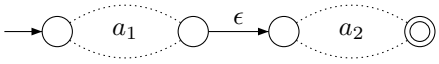
Étant donnés deux automates de Thompson a_1 et a_2 reconnaissant respectivement les langages L_1 et L_2, on peut construire un automate de Thompson reconnaissant le langage $L_1 \cup L_2$ de la manière suivante :



▶ **Question 5** *Écrivez une fonction* th_or *implémentant la construction décrite ci-dessus.*

val th_or: int −> thompson −> thompson −> thompson ∗ int

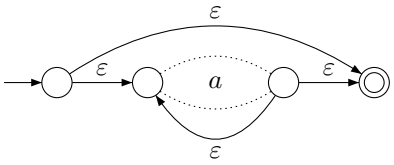
De même, on peut construire un automate de Thompson reconnaissant le langage $L_1 \cdot L_2$ comme suit :



▶ **Question 6** *Écrivez une fonction* th_seq *implémentant la construction décrite ci-dessus.*

val th_seq: int −> thompson −> thompson −> thompson ∗ int

Enfin, étant donné un automate de Thompson a reconnaissant le langage L, l'automate suivant reconnaît le langage L^* :



▶ **Question 7** *Écrivez la fonction* th_star *correspondante.*

val th_star: int −> thompson −> thompson ∗ int

▶ **Question 8** *Déduisez-en une fonction* thompson *prenant en argument un expression rationnelle r et retournant un automate non-déterministe reconnaissant le langage dénoté par r.*

val thompson: regexp −> thompson

1.3 Suppression des ε-transitions

On cherche maintenant à enlever les ε-transitions de l'automate obtenus. L'ensemble d'états initial de l'automate sans ε-transitions est l'ensemble des états qu'on peut atteindre en suivant la chaîne vide, c'est-à-dire l'état initial de l'automate de départ, plus tous les états qu'on peut atteindre à partir de l'état initial en suivant uniquement des ϵ-transitions. De même un état est final si un des états $s_1, \ldots, s_n$ que l'on peut atteindre en ne suivant que des ε-transitions est final.
Nous représenterons les ensembles d'états (d'automates non-déterministes) simplement par des listes de type int list (nous supposerons qu'un même état apparaît au plus une fois dans une telle liste et que celle-ci est ordonnée).

▶ **Question 9** *Écrivez les fonctions* mem_state *qui teste si un état apparaît dans une liste d'états et* insert *qui insere un élement dans une liste trié. Écrivez également une fonction qui renvoie l'ensemble des états d'un automate.*

val mem_state: int −> int list −> bool
val insert : 'a −> 'a list −> 'a list
val states : thompson −> int list

Étant donné un ensemble S d'états d'un automate non-déterministe, on définit son ε-fermeture comme l'ensemble des états accessibles à partir d'un état de S en traversant zéro, une ou plusieurs ε-transitions.

▶ **Question 10** *Définissez une fonction* epsilon_closure *qui calcule la ε-fermeture d'un d'état (d'un automate non-déterministe).*

val epsilon_closure: thompson −> int −> int list

▶ **Question 11** *Déduisez-en une fonction* remove_eps_trans *qui construit un automate sans ε-transitions reconnaissant le même langage qu'un automate donnée. Cet automate peut par exemple être du type que vous avez utilisé dans les sujets précedents.*

val remove_eps_trans: thompson −> int automate

Compilation d'expressions rationnelles

Un corrigé

▶ Question 1

```
type regexp =
  | Epsilon
  | Char of char
  | Or of regexp * regexp
  | Seq of regexp * regexp
  | Star of regexp
```

▶ Question 2

```
let example =
  Seq (Seq (Char 'a' , Star ( Or (
    Seq (Char 'b', Char 'c') , Char 'd') ) ) , Char 'e')
```

▶ Question 3

```
let th_epsilon i =
  {initial = i ; final = i + 1 ;
   trans = [] ; eps_trans = [i,i + 1] } , i + 2
```

▶ Question 4

```
let th_char i c =
  { initial = i ; final = i + 1 ;
    trans = [i,c,i + 1] ; eps_trans = [] } , i + 2
```

▶ Question 5

```
let th_or i a b =
  {
    initial = i ;
    final = i + 1 ;
    trans = List.rev_append a.trans b.trans ;
    eps_trans =
      (i , a.initial)
      :: (i , b.initial)
      :: (a.final , i + 1)
      :: (b.final , i + 1)
      :: (List.rev_append a.eps_trans b.eps_trans)
  } , i + 2
```

▶ Question 6

```
let th_seq i a b =
  {
    initial = a.initial ;
    final = b.final;
    trans = List.rev_append a.trans b.trans ;
    eps_trans = (a.final,b.initial) :: List.rev_append a.eps_trans b.eps_trans ;
  }, i
```

▶ Question 7

```
let th_star i a =
  {
    initial = i ;
    final = i + 1 ;
    trans = a.trans ;
    eps_trans =
      (i,i + 1) :: (i,a.initial) :: (a.final,i + 1) :: (a.final,a.initial)
      :: a.eps_trans ;
  }, i + 2
```

▶ Question 8

```
let thompson r =
  let rec aux i = function
    | Epsilon -> th_epsilon i
    | Char c -> th_char i c
    | Or (a, b) ->
        let x,j = aux i a in
        let y,k = aux j b in
          th_or k x y
    | Seq (a, b) ->
        let x,j = aux i a in
        let y,k = aux j b in
          th_seq k x y
    | Star a ->
        let x,j = aux i a in
          th_star j x
  in fst (aux 0 r)
```

▶ Question 9

```
let mem_state = List.mem

let rec insert x = function
  | [] -> [x]
  | a :: s ->
      if x = a then a :: s
      else if x < a then a :: insert x s
      else x :: a :: s

let states th =
  let st =
    List.fold_left (fun l (s,c,t) -> insert s (insert t l))
      [] th.trans
  in
    List.fold_left (fun l (s,t) -> insert s (insert t l))
      st th.eps_trans
```

▶ Question 10

```
let epsilon_closure th s =
  let rec aux set =
    let new_set =
      List.fold_left (fun l (x,y) ->
                          if appartient x set
                          then insert y l else l)
        set th.eps_trans
    in
      if List.length set = List.length new_set
      then set else aux new_set
  in aux [s]
```

► Question 11

```
type 'a automate =
    {
      i : 'a list ;
      f : 'a list;
      t : ('a * char * 'a) list
    }


let remove_eps_trans th =
  let states = states th in
  let delta s =
    let cl = epsilon_closure th s in
      List.fold_left
(fun l (t,c,v) ->
   if mem_stat t cl then (s,c,v) :: l else l) [] th.trans
  in
  let new_trans =
    List.fold_left
      (fun l s -> List.rev_append (delta s) l)
      [] states
  in
  let new_final =
    List.fold_left
      (fun l s ->
 if mem_stat th.final (epsilon_closure th s)
 then s :: l else l) [] states
  in
    {
      i = epsilon_closure th th.initial;
      f = new_final;
      t = new_trans
    }
```