

MPI\* Info - TP2

## Expressions rationnelles et automates



# 1 Expressions régulières

**Question 1.** Définissez par induction structurale une fonction des expressions rationnelles qui détermine si un langage est vide.

**Corrigé:** On a

- Le langage "vide" est vide
- Soient  $L_1, L_2$  deux expressions rationnelles :
  - $L_1 L_2$  est vide ssi  $L_1$  ou  $L_2$  est vide
  - $L_1 + L_2$  est vide ssi  $L_1$  et  $L_2$  sont vides

```
1 type expr =  
2   Vide  
3   | Epsilon  
4   | Lettre of char  
5   | Union of expr*expr  
6   | Concat of expr*expr  
7   | Etoile of expr
```

**Question 2.** Écrire la fonction `vide : expr -> bool` qui calcule la fonction précédente.

**Corrigé:**

```
1 let rec vide (e : expr) : bool =  
2   match e with  
3   | Vide -> true  
4   | Union(e1,e2) -> vide e1 && vide e2  
5   | Concat(e1,e2) -> vide e1 || vide e2  
6   | _ -> false
```

**Question 3.** Écrire une fonction `a_eps : expr -> bool` telle que `a_eps e` s'évalue à vrai si et seulement si le langage dénoté par l'expression régulière représentée par `e` contient le mot vide  $\varepsilon$ .

**Corrigé:**

```
1 let rec a_eps (e : expr) : bool =  
2   match e with  
3   | Vide -> false  
4   | Epsilon -> true  
5   | Union(e1,e2) -> a_eps e1 || a_eps e2  
6   | Concat(e1,e2) -> a_eps e1 && a_eps e2  
7   | Etoile(e) -> true  
8   | _ -> false
```

**Question 4.** Écrire une fonction `est_eps : expr -> bool` telle que l'appel `est_eps e` s'évalue à vrai ssi le langage dénoté par `e` est exactement  $\{\varepsilon\}$ , le langage formé uniquement du mot vide.

**Corrigé:**

```
1 let rec est_eps (e : expr) : bool =  
2   match e with  
3   | Vide -> false  
4   | Epsilon -> true  
5   | Union(e1,e2) -> est_eps e1 && est_eps e2  
6   | Concat(e1,e2) -> est_eps e1 && est_eps e2  
7   | Etoile(e) -> est_eps e || vide e  
8   | _ -> false
```

**Question 5.** Définissez par induction structurelle une fonction des expressions rationnelles vers les entiers qui calcule la longueur du plus court mot de  $L$  si le langage n'est pas vide, et  $\infty$  sinon.

**Corrigé :** On a

- si  $L$  est vide :  $N(L) = \infty$
- sinon :
  - si  $L = \{\varepsilon\}$  :  $N(L) = 0$
  - si  $L = \{\alpha\}$  avec  $\alpha \in \Sigma$  :  $N(L) = 1$
  - si  $L = (L')^*$  :  $N(L) = 0$
  - soient  $L_1, L_2$  deux expressions qui composent  $L$  :
    - si  $L = L_1 L_2$  :  $N(L) = N(L_1) + N(L_2)$
    - si  $L = L_1 + L_2$  :  $N(L) = \min(N(L_1), N(L_2))$

**Question 6.** Écrire la fonction `longueur_mot_min : expr -> int option` qui calcule la fonction précédente.

**Corrigé :**

```

1 let longueur_mot_min (e: expr) : int option =
2   if vide e then None
3   else
4     let min_aux (i1 : int option) (i2 : int option) : int option =
5       match i1,i2 with
6       |None,None -> None
7       |None,_ -> i2
8       |i1,None -> i1
9       |Some k1, Some k2 -> Some (min k1 k2)
10    in
11
12    let add_aux (i1 : int option) (i2 : int option) : int option =
13      match i1,i2 with
14      |None,None -> None
15      |None,_ -> None
16      |i1,None -> None
17      |Some k1, Some k2 -> Some (k1 + k2)
18    in
19
20    let rec aux (e_aux : expr) : int option =
21      match e_aux with
22      |Epsilon -> Some 0
23      |Lettre(e1) -> Some 1
24      |Etoile(e1) -> Some 0
25      |Union(e1,e2) -> min_aux (aux e1) (aux e2)
26      |Concat(e1,e2) -> add_aux (aux e1) (aux e2)
27      |_ -> None
28    in
29    aux e

```

**Question 7.** On considère l'expression régulière  $e_1 = ab^*a$ . Définir cette expression régulière en Caml. Déterminer le langage  $L(e_1)$  dénoté par cette expression régulière.

**Corrigé :**

```

1 let e1 = Concat( Lettre 'a', Concat( Etoile(Lettre 'b'), Lettre 'a'))

```

Le langage  $L(e_1)$  n'est autre que le langage comportant les mots dont les deux extrémités sont obligatoirement des  $a$ , comportant uniquement des  $b$  entre eux deux, au minimum 0  $b$ .

Si  $L \subset \Sigma^*$  est un langage, son résiduel à gauche (on dira simplement résiduel) pour un mot  $u \in \Sigma$  est le langage  $u^{-1}L = \{v \in \Sigma, uv \in L\}$ . Autrement dit, c'est le langage des mots  $v$  qui peuvent compléter le mot  $u$  pour obtenir un mot de  $L$ .

**Question 8.** Montrer que  $u \in L$  si et seulement si  $\varepsilon \in u^{-1}L$ .

**Corrigé :**

$$\begin{aligned} u \in L &\Leftrightarrow u = u\varepsilon \in L \\ &\Leftrightarrow \varepsilon \in u^{-1}L \end{aligned}$$

On va maintenant chercher à écrire une fonction qui détermine si un mot  $u$  est dans un langage  $L$ . Pour un mot  $u \in \Sigma^*$  l'objectif est donc de calculer  $u^{-1}L$  et de savoir si  $\varepsilon$  est dans ce langage pour savoir si  $u \in L$

**Question 9.** Pour  $u = av$  avec  $u, v \in \Sigma^*$  et  $a \in \Sigma$ . Montrer que  $u^{-1}L = v^{-1}(a^{-1}L)$ .

**Corrigé :** Soit  $w \in \Sigma^*$ ,

$$\begin{aligned} w \in u^{-1}L &\Leftrightarrow uw \in L \\ &\Leftrightarrow avw \in L \\ &\Leftrightarrow vw \in a^{-1}L \text{ (par définition)} \\ &\Leftrightarrow w \in v^{-1}(a^{-1}L) \end{aligned}$$

Il suffit donc de lire les lettres de  $u$  une par une et de déterminer successivement les langages résiduels pour aboutir à  $u^{-1}L$ . Par exemple, déterminons si  $aba$  appartient à  $ab^*a = L(e_1)$ .

**Question 10.** Déterminer  $a^{-1}L(e_1)$  ainsi qu'une expression régulière  $e_2$  qui dénote ce langage.

**Corrigé :** Soit  $v \in \Sigma^*$ ,

$$\begin{aligned} v \in a^{-1}L(e_1) &\Leftrightarrow \exists n \in \mathbb{N}, v = b^n a \\ &\Leftrightarrow v \in \{a; ba; bba; \dots\} \end{aligned}$$

D'où

$$e_2 = b^*a$$

**Question 11.** Déterminer  $(ab)^{-1}L = b^{-1}L(e_2)$  ainsi qu'une expression régulière  $e_3$  qui dénote ce langage

**Corrigé :** Soit  $v \in \Sigma^*$ ,

$$\begin{aligned} v \in (ab)^{-1}L &\Leftrightarrow abv \in L \\ &\Leftrightarrow \exists n \in \mathbb{N}, v = b^n a \\ &\Leftrightarrow v \in \{a; ba; bba; \dots\} \end{aligned}$$

D'où

$$e_2 = e_3 = b^*a$$

**Question 12.** Écrire une fonction `residuel : char -> expr -> expr` qui étant donné un caractère `a` et une expression régulière `e` s'évalue en une expression régulière  $\tilde{e}$  qui dénote le langage résiduel  $\tilde{e} = a^{-1}L(e)$ .

**Corrigé :**

```
1 let rec residuel (a : char) (e : expr) : expr =
2   match e with
3   | Vide -> Vide
4   | Epsilon -> Epsilon
5   | Lettre(b) when a = b -> Epsilon
6   | Lettre(b) -> Vide
7   | Concat(e1,e2) -> if a_eps e1 then Union (Concat (residuel a e1, e2), residuel a e2)
8       else Concat (residuel a e1, e2)
9   | Union(e1,e2)-> Union( residuel a e1, residuel a e2)
10  | Etoile(e1) -> Concat( residuel a e1, Etoile(e1))
```

**Question 13.** Écrire une fonction `appartient : char list -> expr -> bool` qui vérifie si un mot représenté par une liste de caractères appartient au langage dénoté par une expression régulière.

**Corrigé :**

```
1 let rec appartient (l : char list)(e : expr) : bool =
2   match l with
3   | [] -> a_eps e
4   | x::xs -> appartient xs (residuel x e)
```

**Question 14.** Écrire de même une fonction `appartient_bis : string -> expr -> bool` qui a le même comportement que la fonction précédente mais avec une représentation des mots par le type `string` de Caml.

**Corrigé :**

```
1 let appartient_bis (s : string) (e : expr) : bool =
2   let len = String.length s in
3   let rec aux (e_aux : expr)(ind : int) : bool =
4     match ind with
5     | ind when ind=len -> a_eps e_aux
6     | ind -> aux (residuel s.[ind] e_aux) (ind+1)
7   in aux e 0
```

## 2 REGEX et GREP

### Question 15.

15.1 Quels sont les mots qui contiennent au moins un w et au moins un q? (On trouve 6 mots)

```
1 grep -E 'w.*q|q.*w' francais.txt
```

15.2 Quels sont les mots qui contiennent au moins 6 fois la lettre i? (On trouve 4 mots)

```
1 grep -E '(i.*){6}' francais.txt
```

15.3 Quels sont les mots qui contiennent au moins 10 voyelles? (On trouve 112 mots de *anticonstitutionnellement* à *technobureaucratiques*)

```
1 grep -E '([aeiou].*){10}' francais.txt
```

15.4 Quels sont les mots qui commencent par un p et contiennent au moins 9 voyelles? (On trouve 28 mots, de *panoramiqueraient* à *psychophysiologiques*)

```
1 grep -E '^p.*([aeiou].*){9}' francais.txt
```

15.5 Quels sont les mots d'exactly 12 lettres ne contenant ni a ni e ni i? (On trouve 9 mots, de *boursouflons* à *turcmongols*)

```
1 grep -E '^[^aei]{12}$' francais.txt
```

15.6 Quels sont les mots dans lesquels chaque groupe consécutif de 2 lettres contient au moins un s? (On trouve on trouve 37 mots, de *as* à *uses*)

```
1 grep -E '^(s.?)*(.s)*$' francais.txt
```

## 3 Automates déterministes

```
1 type automate =  
2 { taille : int ;  
3   initial : int ;  
4   transitions : (char * int) list array ;  
5   final : bool array }
```

**Question 16.** Il existe dans la bibliothèque Caml une fonction `assoc : 'a -> ('a * 'b) list -> 'b` qui gère les listes de couples (appelées aussi listes associatives) : par exemple `assoc 2 [(1,'a'); (2,'b') ; (2,'c') ; (3,'d')]` vaut `'b`. La fonction `assoc` déclenche l'exception `Not found` en cas d'échec. Écrire une telle fonction.

**Corrigé:**

```
1 exception Not_Found  
2  
3 let rec assoc (elt : 'a) (l : ('a * 'b) list) : 'b =  
4   match l with  
5   | [] -> raise Not_Found  
6   | (x,y)::xs when x = elt -> y  
7   | x::xs -> assoc elt xs
```

**Question 17.** Écrire une fonction `calcul_det : string -> automate -> bool` qui étant donné un mot et un automate supposé déterministe, détermine si l'automate accepte ce mot (on pourra utiliser la fonction `assoc` ou bien la fonction que vous venez de coder). Quelle est sa complexité?

Définir l'automate 1 représenté ci-dessous, et vérifier sur les exemples *aa*, *aba* et *bab* sur la fonction `calcul_det`. est correcte.

**Corrigé :**

```
1 let calcul_det (mot : string) (a : automate) : bool =
2   let n = String.length mot in
3   let rec aux (current_state : int) (indice_mot : int) : bool =
4     match indice_mot with
5     | i when i = n -> a.final.(current_state)
6     | _ ->
7       try
8         aux (assoc mot.[indice_mot] a.transitions.(current_state)) (indice_mot + 1)
9       with Not_Found -> false
10  in
11  aux a.initial 0
```

On obtient, pour  $n = |u|$ , une complexité en  $\mathcal{O}(n)$  (on parcourt au plus la longueur du mot une fois, étant donné qu'il s'agit d'un AFD).

```
1 let a1 = {
2   taille = 5;
3   initial = 0;
4   transitions = [| [('a',1); ('b',2)] ; [('a',1)] ; [('a',3) ; ('b',4)] ; [('b',4)] ; [('a',3)] |];
5   final = [|false; true; false;true;false|]
6 }
```

**Question 18.** Écrire une fonction qui fait un parcours en profondeur de l'automate depuis l'état initial et se contente d'afficher les noms des états parcourus.

**Corrigé :**

```
1 let parcours_profondeur (a : automate) : unit =
2   let len = a.taille in
3   let vus = Array.make len false in
4
5   let rec pp_rec (current_state : int) : unit =
6     if not vus.(current_state) then
7       begin
8         vus.(current_state) <- true ;
9         print_int current_state ; print_string " - " ;
10        List.iter (fun (x,y) -> pp_rec y) a.transitions.(current_state)
11      end
12  in
13
14  for i = 0 to len - 1 do
15    if not vus.(i) then pp_rec i
16  done
```

**Question 19.** Écrire une fonction `accessible` qui supprime les états inaccessibles d'un automate. Pour cela, on doit renuméroter les états, on pourra maintenir deux tableaux `tab_conversion` et `tab_inversion` qui gèrent la correspondance entre nouveaux et anciens états. On adaptera le code de la question précédente pour parcourir les sommets depuis l'état initial en les renumérotant.

**Corrigé :**

```

1 let accessible (a: automate) : automate =
2   let len = a.taille in
3   let vus = Array.make len false in
4   print_string " check 0" ;
5   let rec pp_rec (current_state : int) : unit =
6     if not vus.(current_state) then
7       begin
8         vus.(current_state) <- true ;
9         List.iter (fun (x,y) -> pp_rec y) a.transitions.(current_state)
10      end
11   in
12
13   pp_rec a.initial;
14
15   (* creation du tableau de conversion *)
16
17   let len_accessibles = ref 0 in
18   Array.iter (fun state -> if state then incr(len_accessibles)) vus ;
19   let conversion_tab = Array.make len (-1) in
20
21
22   let ind_tmp = ref 0 in
23   for i = 0 to len -1 do
24     if vus.(i) then
25       begin
26         conversion_tab.(i) <- !ind_tmp;
27         incr(ind_tmp)
28       end
29   done;
30
31   (* On filtre les donnees inutiles de l'ancien automate et on convertit pour creer les nouvelles donnees*)
32
33   let f_filter x =
34     let state = snd(x) in
35     if vus.(state) then
36       begin
37         Some (fst(x), conversion_tab.(state))
38       end
39     else None
40   in
41
42   let ind_tmp = ref 0 in
43   let new_transitions = Array.make !len_accessibles [] in
44   let new_final = Array.make !len_accessibles false in
45   let new_initial = conversion_tab.(a.initial) in
46
47   for i = 0 to len - 1 do
48     if vus.(i) then
49       begin
50         new_transitions.(!ind_tmp) <- List.filter_map f_filter (a.transitions.(i));
51         new_final.(!ind_tmp) <- a.final.(i);
52         incr(ind_tmp)
53       end
54   done;
55
56   {taille = !len_accessibles ; initial = new_initial ; transitions = new_transitions ; final = new_final }

```



**Question 20.** Écrire une fonction `est_vide_auto : automate -> bool` qui détermine si le langage de l'automate est vide. Écrire une fonction `longueur_mot_min_auto : automate -> int option` qui calcule la longueur du mot minimal accepté par l'automate.

**Corrigé :**

```

1  exception Non_Vide
2
3  let est_vide_auto (a : automate) : bool =
4      if a.final.(a.initial) then false
5      else
6          begin
7              let len = a.taille in
8              let vus = Array.make len false in
9
10             let rec pp_rec (current_state : int) : unit =
11                 if not vus.(current_state) then
12                     begin
13                         vus.(current_state) <- true ;
14                         List.iter (fun (x,y) -> pp_rec y) a.transitions.(current_state)
15                     end
16             in
17
18             pp_rec a.initial ;
19
20             try
21                 for i = 0 to len -1 do
22                     if a.final.(i) && vus.(i) then raise Non_Vide
23                 done;
24                 true
25             with Non_Vide -> false
26
27         end
28
29
30  let longueur_min_auto (a : automate) : int option =
31      if est_vide_auto a then None
32      else
33          let len = a.taille in
34          let vus = Array.make len false in
35
36          let rec construit_aux (current_state : int) : int =
37              if not vus.(current_state) then
38                  begin
39                      if a.final.(current_state) then 0
40                      else
41                          let res_mini = ref max_int in
42                          List.iter (fun (x,y) -> let tmp = construit_aux y in if tmp < !res_mini then res_mini := tmp) a.
                              transitions.(current_state) ;
43                          1 + !res_mini
44                      end
45                  else max_int
46          in
47
48          Some (construit_aux a.initial)

```

**Question 21.** Écrire une fonction `langage_auto` qui retourne la liste de tous les mots de taille minimale reconnu par l'automate. Quelle est sa complexité?

**Corrigé:**

```
1 let langage_auto (a : automate) : string list =
2   match longueur_min_auto a with
3   | None -> []
4   | Some 0 -> []
5   | Some longueur_min ->
6     let rec construit_liste_chemin (count : int) (current_state : int) (chemin_aux : (char * int) list) :
7       (char * int) list list =
8       match count with
9       | count when count > longueur_min -> []
10      | count when count = longueur_min ->
11        if a.final.(current_state) then [chemin_aux]
12        else [] (* le chemin ne mene nulle part *)
13      | _ -> applique_liste a.transitions.(current_state) count current_state chemin_aux
14
15    and applique_liste (l_state : (char * int) list) (count : int) (current_state : int) (chemin_aux : (
16      char * int) list) : (char * int) list list =
17      match l_state with
18      | [] -> []
19      | (lettre, state)::xs ->
20        construit_liste_chemin (count + 1) state ((lettre, current_state)::chemin_aux)
21        @ applique_liste xs count current_state chemin_aux
22    in
23
24    let chemins_min = construit_liste_chemin 0 a.initial [] in
25
26    let rec lecture_chemin (l_chemin : (char * int) list) : string =
27      match l_chemin with
28      | [] -> ""
29      | (lettre, state)::xs -> (lecture_chemin xs) ^ (String.make 1 lettre)
30    in
31
32    let rec lecture_chemin (l_chemin : (char * int) list list) : string list =
33      match l_chemin with
34      | [] -> []
35      | l::ls -> (lecture_chemin l)::(lecture_chemin ls)
36    in
37
38    lecture_chemin chemins_min
```

