MPI* Info - TP2

Expressions rationnelles et automates

O. Caffier

1 Expressions régulières

Question 1. Définissez par induction structurelle 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

```
type expr =
    Vide
    |Epsilon
    |Lettre of char
    |Union of expr*expr
    |Concat of expr*expr
    |Etoile of expr
```

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

```
let rec vide (e : expr) : bool =
  match e with
  |Vide -> true
  |Union(e1,e2) -> vide e1 && vide e2
  |Concat(e1,e2) -> vide e1 || vide e2
  |_ -> 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 ε.
Corrigé :

```
let rec a_eps (e : expr) : bool =
  match e with
  |Vide -> false
  |Epsilon -> true
  |Union(e1,e2) -> a_eps e1 || a_eps e2
  |Concat(e1,e2) -> a_eps e1 && a_eps e2
  |Etoile(e) -> true
  |_ -> 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 :

```
let rec est_eps (e : expr) : bool =
  match e with
  |Vide -> false
  |Epsilon -> true
  |Union(e1,e2) -> est_eps e1 && est_eps e2
  |Concat(e1,e2) -> est_eps e1 && est_eps e2
  |Etoile(e) -> est_eps e || vide e
  |_ -> 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 ∞ 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 :

```
let longueur_mot_min (e: expr) : int option =
  if vide e then None
  else
    let min_aux (i1 : int option) (i2 : int option) : int option =
      match i1,i2 with
      |None,None -> None
      |None, _ -> i2
      |i1,None -> i1
      |Some k1, Some k2 -> Some (min k1 k2)
    in

    let add_aux (i1 : int option) (i2 : int option) : int option =
      match i1,i2 with
      |None,None -> None
      |None, _ -> None
      |i1,None -> None
      |Some k1, Some k2 -> Some (k1 + k2)
    in

    let rec aux (e_aux : expr) : int option =
      match e_aux with
      |Epsilon -> Some 0
      |Lettre(e1) -> Some 1
      |Etoile(e1) -> Some 0
      |Union(e1,e2) -> min_aux (aux e1) (aux e2)
      |Concat(e1,e2) -> add_aux (aux e1) (aux e2)
      |_ -> None
    in
    aux e
```

Question 7. On considère l'expression régulière $e1 = ab^*a$. Définir cette expression régulière en Caml. Déterminer le langage $L(e_1)$ dénoté par cette expression régulière.
Corrigé :

```
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 ε 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 $v \in \Sigma^*$,

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

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 $v \in Sigma^*$,

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

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 :

```
let rec residuel (a : char) (e : expr) : expr =
  match e with
  |Vide -> Vide
  |Epsilon -> Epsilon
  |Lettre(b) when a = b -> Epsilon
  |Lettre(b) -> Vide
  |Concat(e1,e2) -> if a_eps e1 then Union (Concat (residuel a e1, e2), residuel a e2)
      else Concat (residuel a e1, e2)
  |Union(e1,e2)-> Union( residuel a e1, residuel a e2)
  |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 :

```
let rec appartient (l : char list)(e : expr) : bool =
  match l with
  |[] -> a_eps e
  |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 :

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

```
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)

```
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*)

```
grep -E '([aeiouy].*){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*)

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

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

```
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*)

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

3 Automates déterministes

```
type automate =
  { taille : int ;
    initial : int ;
    transitions : (char * int) list array ;
    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 :

```
exception Not_Found

let rec assoc (elt : 'a) (l : ('a * 'b) list) : 'b =
  match l with
  |[] -> raise Not_Found
  |(x,y)::xs when x = elt -> y
  |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 :

```
let calcul_det (mot : string) (a : automate) : bool =
  let n = String.length mot in
  let rec aux (current_state : int) (indice_mot : int) : bool =
    match indice_mot with
    |i when i = n -> a.final.(current_state)
    |_ ->
        try
          aux (assoc mot.[indice_mot] a.transitions.(current_state)) (indice_mot +1)
        with Not_Found -> false
  in
  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).

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

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 :

```
let parcours_profondeur (a : automate) : unit =
  let len = a.taille in
  let vus = Array.make len false in

  let rec pp_rec (current_state : int) : unit =
    if not vus.(current_state) then
      begin
        vus.(current_state) <- true ;
        print_int current_state ; print_string " - " ;
        List.iter (fun (x,y) -> pp_rec y) a.transitions.(current_state)
      end
  in

  for i = 0 to len -1 do
    if not vus.(i) then pp_rec i
  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 :

```
let accessible (a: automate) : automate =
  let len = a.taille in
  let vus = Array.make len false in
  print_string " check 0" ;
  let rec pp_rec (current_state : int) : unit =
    if not vus.(current_state) then
      begin
        vus.(current_state) <- true ;
        List.iter (fun (x,y) -> pp_rec y) a.transitions.(current_state)
      end
  in

  pp_rec a.initial;

  (* creation du tableau de conversion *)

  let len_accessibles = ref 0 in
  Array.iter (fun state -> if state then incr(len_accessibles)) vus ;
  let conversion_tab = Array.make len (-1) in


  let ind_tmp = ref 0 in
  for i = 0 to len -1 do
    if vus.(i) then
      begin
        conversion_tab.(i) <- !ind_tmp;
        incr(ind_tmp)
      end
  done;

  (* On filtre les donnees inutiles de l'ancien automate et on convertit pour creer les nouvelles donnees*)

  let f_filter x =
    let state = snd(x) in
    if vus.(state) then
      begin
        Some (fst(x), conversion_tab.(state))
      end
    else None
  in

  let ind_tmp = ref 0 in
  let new_transitions = Array.make !len_accessibles [] in
  let new_final = Array.make !len_accessibles false in
  let new_initial = conversion_tab.(a.initial) in

  for i = 0 to len - 1 do
    if vus.(i) then
      begin
        new_transitions.(!ind_tmp) <- List.filter_map f_filter (a.transitions.(i));
        new_final.(!ind_tmp) <- a.final.(i);
        incr(ind_tmp)
      end
  done;

  {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.

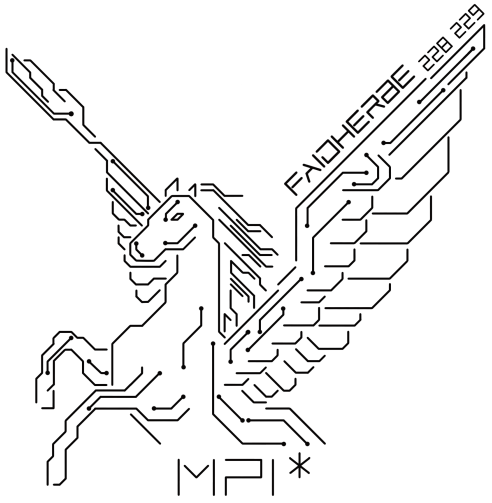
Corrigé :

```
exception Non_Vide

let est_vide_auto (a : automate) : bool =
  if a.final.(a.initial) then false
  else
    begin
      let len = a.taille in
      let vus = Array.make len false in

      let rec pp_rec (current_state : int) : unit =
        if not vus.(current_state) then
          begin
            vus.(current_state) <- true ;
            List.iter (fun (x,y) -> pp_rec y) a.transitions.(current_state)
          end
      in

      pp_rec a.initial ;

      try
        for i = 0 to len -1 do
          if a.final.(i) && vus.(i) then raise Non_Vide
        done;
        true
      with Non_Vide -> false

    end


let longueur_min_auto (a : automate) : int option =
  if est_vide_auto a then None
  else
    let len = a.taille in
    let vus = Array.make len false in

    let rec construit_aux (current_state : int) : int =
      if not vus.(current_state) then
        begin
          if a.final.(current_state) then 0
          else
            let res_mini = ref max_int in
            List.iter (fun (x,y) -> let tmp = construit_aux y in if tmp < !res_mini then res_mini := tmp) a.
                transitions.(current_state) ;
            1 + !res_mini
        end
      else max_int
    in

    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 :

```
let langage_auto (a : automate) : string list =
  match longueur_min_auto a with
  |None -> []
  |Some 0 -> []
  |Some longueur_min ->
      let rec construit_liste_chemins (count : int) (current_state : int) (chemin_aux : (char * int) list) :
           (char * int) list list =
        match count with
        |count when count > longueur_min -> []
        |count when count = longueur_min  ->
            if a.final.(current_state) then [chemin_aux]
            else [] (* le chemin ne mene nulle part *)
        |_ -> applique_liste a.transitions.(current_state) count current_state chemin_aux

      and applique_liste (l_state : (char * int) list) (count : int) (current_state : int) (chemin_aux : (
          char * int) list) : (char * int) list list =
        match l_state with
        |[] -> []
        |(lettre,state)::xs ->
          construit_liste_chemins (count + 1) state ((lettre,current_state)::chemin_aux)
          @ applique_liste xs count current_state chemin_aux
      in

      let chemins_min = construit_liste_chemins 0 a.initial [] in

      let rec lecture_chemin (l_chemin : (char * int) list ) : string =
        match l_chemin with
        |[] -> ""
        |(lettre, state)::xs -> (lecture_chemin xs) ^ (String.make 1 lettre)
      in

      let rec lecture_chemins (l_chemins : (char * int) list list) : string list =
        match l_chemins with
        |[] -> []
        |l::ls -> (lecture_chemin l):: (lecture_chemins ls)
      in

      lecture_chemins chemins_min
```


FAIDHERBE 228 229
MPI*