Correction de l'épreuve d'informatique Mines 2007

I. Expressions rationnelles

Question 1 Il est clair que $(\varepsilon+\varepsilon)$ et ε sont équivalentes, puisque ces deux expressions décrivent le langage $\{\varepsilon\}$. On a de même équivalence entre $(e_1+\emptyset)$ et e_1, entre $(\varepsilon\cdot\emptyset)$ et $\emptyset$, entre $(e_1\cdot\emptyset)$ et $\emptyset$, entre $(e_1\cdot\varepsilon)$ et e_1, entre $((e_1+\varepsilon)+(e_2+\varepsilon))$ et $(e_1+e_2)+\varepsilon$, entre $(e_1\cdot(e_2+\varepsilon))$ et $(e_1\cdot e_2)+e_1$ et enfin entre $((e_1+\varepsilon)\cdot(e_2+\varepsilon))$ et $((e_1\cdot e_2)+(e_1+e_2))+\varepsilon$.

Question 2 On montre simplement ce résultat par induction structurelle. À chaque fois, les expressions régulières e et e' sont sans $\emptyset$ ni ε.

– **Somme** Du fait de la commutativité et de l'associativité de la somme, il suffit de considérer les cas suivants :

$$(\emptyset+\emptyset)\sim\emptyset \quad (\varepsilon+\emptyset)\sim\varepsilon \quad (e+\emptyset)\sim e \quad ((e+\varepsilon)+\emptyset)\sim e+\varepsilon$$
$$(\varepsilon+\varepsilon)\sim\varepsilon \quad (e+\varepsilon)\sim e+\varepsilon \quad ((e+\varepsilon)+\varepsilon)\sim e+\varepsilon$$
$$(e+e')\sim(e+e') \quad (e+(e'+\varepsilon))\sim(e+e')+\varepsilon \quad ((e+\varepsilon)+(e'+\varepsilon))\sim(e+e')+\varepsilon$$

– **Concaténation** Le nombre de résultats à considérer semble plus important que pour la somme, puisque la concaténation n'est pas commutative. Cependant, il faut remarquer que $\emptyset$ est absorbant pour la concaténation à gauche comme à droite, alors que ε est élément neutre. Enfin, la concaténation est distributive sur la somme. Il en découle donc directement que la concaténation de deux expressions rationnelles peut être mise sous bonne forme.

– **Étoile** On a $\emptyset^\star\sim\varepsilon$, $\varepsilon^\star\sim\varepsilon$, $e^\star\sim(e\cdot e^\star)+\varepsilon$ et $(e+\varepsilon)^\star\sim(e\cdot e^\star)+\varepsilon$.

On vérifie à chaque fois que les expressions obtenues à droite ont bien l'une des formes possibles.

Question 3 Si l'on est sûr que l'entier code soit un symbole, soit un caractère, on peut être tenté d'écrire :

```
let est_symbole n = n < 0 ;;
```

Cependant, il faut impérativement utiliser les constantes indiquées (car cela assure la compatibilité même si l'on décide plus tard de représenter les expressions autrement).

```
let est_symbole s =
 (s = ETOILE) || (s = PLUS) || (s = POINT) || (s = P_O) || (s = P_F)
;;
```

Question 4 On propose la version fonctionnelle suivante, où dans la fonction auxiliaire, `i` désigne la position et `par` le nombre de parenthèses ouvertes mais non fermées :

```
let cesure expr debut fin =
  let rec aux i par =
    if i = fin then -1 else
    match expr.(i) with
      n when n = PLUS -> if (par = 0) then i else aux (i+1) par
    | n when n = POINT -> if (par = 0) then i else aux (i+1) par
    | n when n = P_O -> aux (i+1) (par+1)
    | n when n = P_F -> aux (i+1) (par-1)
    | _ -> aux (i+1) par
  in
  aux (debut+1) 0
;;
```

Même si ce n'est pas indiqué, il est clair que `debut` et `fin` sont supposés être tels que on ne pourra fermer que des parenthèses que l'on a ouvertes (autrement dit, on a toujours `par` ≥ 0). Sinon, il est facile d'ajouter un test lorsque l'on lit une parenthèse fermante.

On remarquera de plus la lourdeur du code conséquence de l'utilisation de constantes et non d'un type somme comme par exemple :

```
type 'a symbole = Etoile | Plus | Point | P_O | P_F | Lettre of 'a ;;
```

Question 5 Il s'agit de vérifier que l'expression donnée est *bien formée*, pour reprendre le terme usuel. D'après l'énoncé, une telle expression est, puisque l'on excepte $\emptyset$ et ε, soit une lettre, soit une expression de la forme $e^\star$, (e_1+e_2) ou $(e_1\cdot e_2)$ avec e, e_1 et e_2 bien formée. On en déduit directement la fonction demandée, en remarquant toutefois qu'il faut commencer par repérer une étoile éventuelle :

```
let rec est_rationnelle expr debut fin =
  if expr.(fin) = ETOILE
  then (* on cherche une expression de la forme e* *)
    expr.(debut) = P_O && expr.(fin - 1) = P_F
    && est_rationnelle expr (debut + 1) (fin - 2)
  else
    if debut = fin && not (est_symbole expr.(debut))
    then true (* on a une lettre *)
    else
      if expr.(debut) = P_O && expr.(fin) = P_F
      then (* notre expression est de la forme (...) *)
```

```
          (* il faut alors imperativement qu'entre les deux parentheses *)
          (* on trouve une expression rationnelle, puis PLUS ou POINT,  *)
          (* puis une autre expression rationnelle. *)
      begin
        let ces = cesure expr debut fin in
        if ces = -1 then false
        else (est_rationnelle expr (debut + 1) (ces - 1)) &&
             (est_rationnelle expr (ces + 1) (fin - 1))
      end
      else false
;;
```

Question 6 En notant $\mathscr{L}(e)$ le langage décrit par l'expression rationnelle e, on a :
– si $a \in \Sigma$, alors $\varepsilon \notin \mathscr{L}(a)$;
– pour toute expression rationnelle e, $\varepsilon \in \mathscr{L}(e^\star)$;
– $\varepsilon \in \mathscr{L}(e_1 + e_2) \Leftrightarrow (\varepsilon \in \mathscr{L}(e_1)$ **ou** $\varepsilon \in \mathscr{L}(e_2))$;
– $\varepsilon \in \mathscr{L}(e_1 \cdot e_2) \Leftrightarrow (\varepsilon \in \mathscr{L}(e_1)$ **et** $\varepsilon \in \mathscr{L}(e_2))$.
Cela décrit directement un algorithme récursif, les cas "de base" étant les lettres et les étoiles, et les règles pour une concaténation et une somme étant données par les équivalences ci-dessus.

Question 7 On reprend la structure de la fonction `est_rationnelle`

```
let rec expression_vers_arbre expr debut fin =
  if expr.(fin) = ETOILE
  then
    if expr.(debut) = P_O && expr.(fin - 1) = P_F
    then Unaire_ETOILE (expression_vers_arbre expr debut (fin - 2))
    else failwith "Pas une expression rationnelle"
  else
    if expr.(debut) = expr.(fin) && not (est_symbole expr.(debut))
    then Feuille (expr.(debut))
    else
      if expr.(debut) = P_O && expr.(fin) = P_F
      then begin
        let ces = cesure expr debut fin in
        if ces = -1 then failwith "Pas une expression rationnelle"
        else
          let a_1 = expression_vers_arbre expr (debut + 1) (ces - 1)
          and a_2 = expression_vers_arbre expr (ces + 1) (fin - 1) in
          if expr.(ces) = POINT
          then Binaire_POINT (a_1, a_2)
          else Binaire_PLUS (a_1, a_2)
      end
      else failwith "Pas une expression rationnelle"
;;
```

Question 8 L'algorithme esquissé à la question 6 se traduit très facilement à l'aide d'un arbre :

```
let rec contient_epsilon = function
    Feuille _ -> false
  | Unaire_ETOILE _ -> true
  | Binaire_PLUS (a_1, a_2) ->
      (contient_epsilon a_1) || (contient_epsilon a_2)
  | Binaire_POINT (a_1, a_2) ->
      (contient_epsilon a_1) && (contient_epsilon a_2)
;;
```

II. Langages locaux

Question 9 Pour $(\mathtt{a} \cdot (\mathtt{a})^\star)$, il s'agit d'éviter les b. Ainsi, on a $I = F = \{\mathtt{a}\}$ et $P = \{\mathtt{aa}\}$ et le langage ne contient pas le mot vide donc $\alpha =$ *faux*.

Pour $((\mathtt{a} \cdot \mathtt{b}))^\star$, on utilisera le quadruplet :

$$(\{\mathtt{a}\}, \{\mathtt{b}\}, \{\mathtt{ab}, \mathtt{ba}\}, \textit{vrai})$$

Question 10 Considérons chaque problème proposé :
– **Problème de l'union** : L'union de deux langages locaux n'est pas forcément locale.
Ainsi, les langages $((\mathtt{a} \cdot \mathtt{b}))^\star$ et $((\mathtt{b} \cdot \mathtt{a}))^\star$ sont locaux mais leur réunion ne l'est pas. En effet, dans les deux cas, les facteurs de longueur 2 permis sont $\{\mathtt{ab}, \mathtt{ba}\}$, il en est donc de même pour l'union. Les lettres initiales permises pour la réunion comme les lettres finales sont a et b. Mais dans ce cas, le mot aba est permis alors qu'il n'est pas dans la réunion considérée.
– **Problème de l'intersection** : L'intersection de deux langages locaux est bien locale.
Pour trouver les lettres initiales, les lettres finales et les facteurs de longueur 2 permis, il suffit d'effectuer l'intersection des ensembles correspondants pour chacun des langages. De plus, la présence éventuelle de mot vide ne pose pas non plus de problème.
– **Problème de la concaténation** : La concaténation de deux langages locaux n'est pas forcément locale.
En effet, reprenons l'exemple de $((\mathtt{a} \cdot \mathtt{b}))^\star$ et $((\mathtt{b} \cdot \mathtt{a}))^\star$. Si leur concaténation était locale, seule a serait permise comme lettre initiale et finale. Par contre, les facteurs de

longueur 2 autorisée seraient ab, ba, bb, le facteur bb apparaissant avec la concaténation. Mais alors ce langage contiendrait le mot abbba que ne peut être obtenu en concaténant des mots des langages considérés.

– **Problème de l'étoile** : L'étoile d'un langage local est aussi locale.
En plus d'indiquer que le mot vide est accepté, on remarque que pour les mots non vides, les lettres initiales et finales permises restent les mêmes, et que de nouveaux facteurs de longueurs 2 sont autorisés : ceux constitués d'une lettre final puis d'une lettre initiale. Ainsi, du triplet (I, F, P, α), on passe au triplet $(I, F, P \cup F \times I, \mathit{vrai})$.

Question 11 Le langage des mots dont la lettre initiale est dans I est $I \cdot \Sigma^\star$. Pour les lettres finales, il faut considérer $\Sigma^\star \cdot F$.
Le langage des mots de longueur 2 qui ne sont pas dans P est $\Sigma^2 \cap P^c$.
Le langage des mots contenant au moins un facteur de longueur 2 qui n'est pas dans P est $\Sigma^\star \cdot \left(\Sigma^2 \cap P^c\right) \cdot \Sigma^\star$.
Enfin, le langage $L \setminus \{\varepsilon\}$ est $L \cap (\Sigma \cdot \Sigma^\star)$ que l'on peut noter $L \cap \Sigma^+$.

Question 12 On en déduit qu'un langage local L est rationnel, puisque $L \setminus \{\varepsilon\}$ n'est aussi. En effet, un mot non vide de L est un mot dont l'initiale est dans I, la finale est dans F et qui ne contient pas de facteur de longueur 2 n'appartenant pas à P. Autrement dit :

$$L \setminus \{\varepsilon\} = (I \cdot \Sigma) \cap (\Sigma \cdot F) \cap \left(\Sigma^\star \cdot \left(\Sigma^2 \cap P^c\right) \cdot \Sigma^\star\right)^c$$

Question 13 On a par exemple :

```
let appartient mot i f p =
  let l = vect_length mot in
  i.(mot.(0)) && f.(mot.(l - 1)) && (
    let pos = ref 0 in
    while (!pos < (l-1) && p.(mot.(!pos)).(mot.(!pos + 1))) do
      pos := !pos + 1
    done ;
    !pos = (l-1)
  )
;;
```

III. Mots appartenant au langage décrit par une expression rationnelle

Question 14 D'après la question 10 et la réponse positive au problème de l'étoile, il suffit pour montrer ce résultat par induction structurelle de se limiter aux cas de la somme et de la concaténation.

Soient donc L_1 et L_2 deux langages locaux décrits respectivement par $(I_1, F_1, P_1, \alpha_1)$ et $(I_2, F_2, P_2, \alpha_2)$ sur deux alphabets disjoints (ou, de façon équivalentes, deux langages décrits par des expressions rationnelles n'ayant pas de lettre en commun).

Montrons tout d'abord que $L_1 + L_2$ est local, et qu'il est décrit par le quadruplet $(I_1 \cup I_2, F_1 \cup F_2, P_1 \cup P_2, \alpha_1 \vee \alpha_2)$: un mot de $L_1 + L_2$ commençant par une lettre dans I_1 (resp. I_2) aura forcément tous ses facteurs de longueur 2 dans P_1 (resp. P_2) et finira par une lettre de F_1 (resp. F_2).

Concernant la concaténation, supposons pour commencer que $\alpha_1 = \alpha_2 = \mathit{faux}$ et montrons que $L_1 \cdot L_2$ est local, et décrit par $(I_1, F_2, P_1 \cup P_2 \cup F_1 \times I_2, \mathit{faux})$. En effet, un mot de $L_1 \cdot L_2$ début par des lettres présentes dans L_1, puis arrive un facteur de longueur 2 non inclus dans P_1. Ce facteur appartient alors à $F_1 \times I_2$, ce qui signifie que la fin du mot appartient à L_2.

On peut facilement adapter la preuve aux autres valeurs des α_i. Ainsi, si $\alpha_1 = \mathit{vrai}$, on a le langage L_2 qui est inclus dans $L_1 \cdot L_2$. Cela se traduit par le fait que l'ensemble des lettres initiales doit alors être $I_1 \cup I_2$.

Question 15 Il est clair que ce langage est décrit par :

$$(\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\}, \{\mathtt{ac}, \mathtt{ca}\}, \mathit{vrai})$$

Question 16 Les remarques précédentes sur la présence ou non du mot vide lors du concaténation permettent d'obtenir facilement la fonction voulue, qui utilisera contient_epsilon :

```
let rec calcul_I arbre i =
  match arbre with
    Feuille f -> i.(f) <- true
  | Unaire_ETOILE arbre' -> calcul_I arbre' i
  | Binaire_PLUS (a_1, a_2) -> begin
      calcul_I a_1 i ;
      calcul_I a_2 i
    end
  | Binaire_POINT (a_1, a_2) -> begin
      calcul_I a_1 i ;
      if (contient_epsilon a_1) then calcul_I a_2 i
    end
;;
```

Question 17 Essayons d'écrire une fonction efficace.

```
let ajout_couples produit t1 t2 =
  let dim = vect_length produit in
```

```
  for i = 0 to dim - 1 do
    if t1.(i) then begin
      for j = 0 to dim - 1 do
        if t2.(j) then produit.(i).(j) <- true
      done
    end
  done
;;
```

Question 18 Étudions tout d'abord quels sont les facteurs de longueur 2 présents dans un langage. On raisonne par cas selon la forme de l'expression rationnelle décrivant le langage.
- si $e = a$ pour $a \in \Sigma$, le langage ne comporte pas de facteur de longueur 2 ;
- si $e = (e')^\star$, les facteurs sont ceux de e', plus les facteurs de la forme ab pour a finale et b initiale dans e' ;
- si $e = (e_1 + e_2)$, on calcule la réunion des deux ensembles de facteurs ;
- si $e = (e_1 \cdot e_2)$, les facteurs sont ceux de e_1, ceux de e_2 ainsi que ceux de la forme ab avec a finale dans e_1 et b initiale dans e_2.

On en déduit la fonction suivante :

```
let rec calcul_P arbre p =
  match arbre with
  Feuille _ -> ()
| Unaire_ETOILE arbre' -> begin
    calcul_P arbre' p ;
    let dim = vect_length p in
    let f = make_vect dim false
    and i = make_vect dim false in
    calcul_F arbre' f ;
    calcul_I arbre' i ;
    ajout_couples p f i
  end
  | Binaire_PLUS (a_1, a_2) -> begin
      calcul_P a_1 p ;
      calcul_P a_2 p
    end
  | Binaire_POINT (a_1, a_2) -> begin
      calcul_P a_1 p ;
      calcul_P a_2 p ;
    let dim = vect_length p in
    let f = make_vect dim false
    and i = make_vect dim false in
    calcul_F a_1 f ;
    calcul_I a_2 i ;
    ajout_couples p f i
    end
;;
```

Question 19 Si toutes les lettres apparaissant dans e sont distinctes, le langage décrit est local. On peut donc tester l'appartenance d'un mot à ce langage à l'aide de la fonction `appartient`, les ensembles I, F et P étant obtenus à l'aide des fonctions précédentes.

Question 20 Pour déterminer si toutes les lettres d'une expression régulière sont distinctes, il suffit de la parcourir en ayant une liste des lettres déjà vues.

```
let lettres_distinctes expr n =
  let present = make_vect n false
  and l = vect_length expr in
  let rec aux pos =
    if pos = l then true
    else
      if est_symbole expr.(pos)
      then aux (pos + 1)
      else if present.(expr.(pos))
        then false
        else begin
          present.(expr.(pos)) <- true ;
          aux (pos + 1)
        end
  in
  aux 0
;;
```

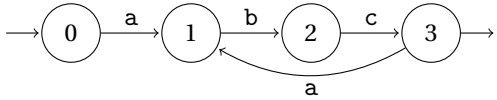
Question 21 D'après la remarque précédente, l'algorithme peut se dérouler ainsi, où l'on teste l'appartenance du mot m au langage décrit par e :

1. On renumérote l'expression e en une expression e' ne comportant que des lettres distinctes. On note ϕ la fonction de traduction des lettres.
2. On construit la fonction d'appartenance au langage local engendré par e', comme décrit dans les questions précédentes.
3. Pour chaque mot m' dont l'image par ϕ est m, on teste l'appartenance de m' au langage local décrit par e' à l'aide de la fonction précédente.

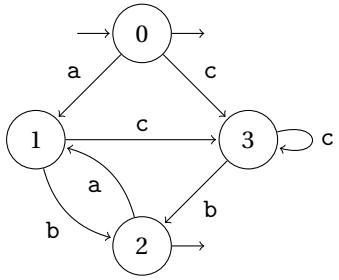
Cet algorithme est cependant d'une complexité désastreuse.

IV. Algorithme de Glushkov

Question 22 L'automate suivant convient :



Question 23 On peut considérer l'automate suivant :



Question 24 D'après les exemples précédents, il semble clair que l'on peut construire un automate de la façon suivante :
- l'ensemble des états Q est composé d'un état *init* ainsi qu'un état q_a pour toutes les lettres a de Σ ;
- seul l'état *init* est initial ;
- les états finaux sont les q_a pour $a \in F$, auquel on ajoute *init* si $\alpha = vrai$;
- on a une transition $init \xrightarrow{a} q_a$ pour toute lettre $a \in I$;
- on a une transition $q_a \xrightarrow{b} q_b$ pour tous les mots ab appartenant à P.

Question 25 En combinant les questions précédentes, étant donné une expression régulière e, on note e' l'expression régulière obtenue à partir de e et ayant toutes ses lettres distinctes, et ϕ la fonction correspondance entre les lettres de chaque expression. Puisque e' correspond à un langage local, on construit l'automate $\mathscr{A}'$ correspondant grâce à l'algorithme précédent. Pour finir, on obtient l'automate $\mathscr{A}$ à partir de $\mathscr{A}'$ en remplaçant toutes les étiquettes par leur image par ϕ.

Un mot m est alors reconnu par $\mathscr{A}$ si, et seulement si, en considérant le chemin réussi correspondant dans $\mathscr{A}$ ainsi que le chemin équivalent dans $\mathscr{A}'$, il existe un mot m' reconnu par $\mathscr{A}$ tel que m est l'image lettre à lettre de m' par ϕ. D'après la remarque faite juste avant la question 21, on en déduit que m est reconnu par $\mathscr{A}$ si, et seulement si il est dans le langage engendré par e.

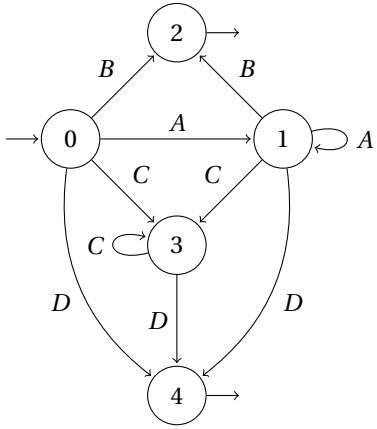
Question 26 On a $expression'_2 = ((A)^\star \cdot (B + ((C)^\star \cdot D)))$ avec :

$$\phi : A \mapsto \texttt{b} \quad B \mapsto \texttt{a} \quad C \mapsto \texttt{a} \quad D \mapsto \texttt{b}$$

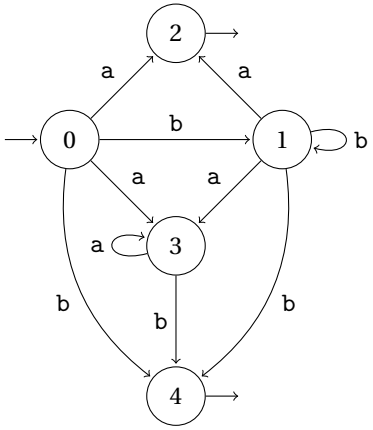
Le langage local engendré par $expression'_2$ est décrit par :

$$I = \{A, B, C, D\} \quad F = \{B, D\} \quad P = \{AA, AB, AC, AD, CC, CD\} \quad \alpha = faux$$

L'automate correspondant est le suivant :



Après renumérotation, on obtient :



Question 27 On déterminise l'automate en considérant des ensembles d'états. Le nouvel état initial est $\{0\}$, qui a une transition étiquetée par a vers $\{2,3\}$, etc.