

Correction

Informatique – DM 2

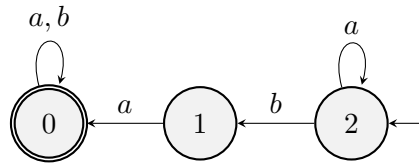
Sujet Centrale 2022

I Mots et automates

Q1. Le langage L_1 est le langage $\{uaba^n \mid u \in \Sigma^*, n \geq 0\}$, aussi donné par l'expression rationnelle Σ^*aba^* .

Son miroir \tilde{L}_1 est le langage $\{a^nba_u \mid u \in \Sigma^*, n \geq 0\}$, donné par l'expression rationnelle $a^*ba\Sigma^*$.

Q2. Voici un automate \tilde{A}_1 qui convient :



Q3. L'automate miroir $\tilde{A} = (Q, I', F', T')$ est donné par :

- $I' = F$
- $F' = I$
- $T' = \{(q, a, q') \mid (q', a, q) \in T\}$

Vérifions que $u \in L_A$ si et seulement si $\tilde{u} \in L_{\tilde{A}}$:

- $\boxed{\Rightarrow}$ Si $u = u_1 \dots u_n \in L_A$, alors il existe un chemin acceptant $q_0 \xrightarrow{u_1} q_1 \dots \xrightarrow{u_n} q_n$ d'étiquette u , avec $q_0 \in I$, $q_n \in F$ et pour tout i , $(q_{i-1}, u_i, q_i) \in T$. Alors par construction de \tilde{A} , on a $q_n \in I'$, $q_0 \in F'$ et pour tout i , $(q_i, u_i, q_{i-1}) \in T'$. Alors le chemin $q_n \xrightarrow{u_n} q_{n-1} \dots \xrightarrow{u_0} q_0$ est un chemin acceptant dans \tilde{A} d'étiquette \tilde{u} . Donc $\tilde{u} \in L_{\tilde{A}}$.
- Avec la définition de \tilde{A} , on remarque que $\tilde{\tilde{A}} = A$. On peut donc se ramener au cas précédent, et obtenir ce qui suit : si $\tilde{u} \in L_{\tilde{A}}$, alors $\tilde{\tilde{u}} \in L_{\tilde{\tilde{A}}} = L_A$

Q4. Voici la fonction demandée.

```

1 let transpose (a : automate) : automate =
2     (* fonction auxiliaire traitant la liste des transitions *)
3     let rec aux_trans delta =
4         match delta with
5         | [] -> acc
6         | (q1,l,q2)::t -> (q2,l,q1)::(aux_trans t)
7     in
8     {nb = a.nb ; init = a.final ; final = a.init ; trans =
      aux_trans a.trans}

```

On peut également appliquer `List.map` à `a.trans` en fournissant une fonction bien choisie, pour s'épargner l'écriture d'une fonction auxiliaire. Voici ce que cela donne :

```

1 let transpose (a : automate) : automate =
2     let aux x =
3         let (q1,a,q2) = x in (q2,a,q1)
4     {nb = a.nb ; init = a.final ; final = a.init ; trans =
      List.map aux a.trans}

```

Q5. La complexité fonction demande d'évaluer la complexité de la création de chacun des champs du nouvel automate :

- La création des champs `nb`, `init`, `final` demande une recopie des valeurs de ces champs dans l'automate `a`. Leurs complexités respectives sont donc $O(1)$, $O(|I|)$, $O(|F|)$.
- La création du champ `trans` demande un parcours de la liste `a.trans` ; à chaque élément de cette liste, on applique une transformation qui s'exécute en temps constant. La création de ce champ a donc une complexité asymptotiquement majorée par $O(|T|)$.
- Au total, le nombre d'opérations nécessaire est majoré par $O(|I| + |F| + |T|)$.

Q6. Voici la fonction demandée.

```

1 let palindrome (w : string) : bool =
2     let i = ref 0 in
3     let res = ref true in
4     let n = String.length s in
5     while (!res) && (!i < n/2) do
6         if s.[!i] <> s.[n-!i] then
7             res := false;
8             i := !i+1
9     done;
10    !res

```

On remarque qu'il suffit, dans la boucle, de ne parcourir que la moitié du mot, et de comparer deux lettres à chaque étape : la i -ème, et la $n - i$ -ème (où n est la longueur du mot).

Si le mot est de longueur impaire, la lettre centrale n'est pas examinée. Au plus, on aura donc fait n accès en mémoire pour examiner les lettres, et $\frac{n}{2}$ comparaisons. La complexité est donc bien linéaire.

Q7. Si Σ ne contient qu'une lettre a , alors $\Sigma^* = \{a^n \mid n \geq 0\}$. On remarque immédiatement que tous ces mots sont des palindrômes, $\text{Pal}(\Sigma) = \Sigma^*$. Ce langage est alors engendré par l'expression rationnelle Σ^* , donc il s'agit d'un langage rationnel.

Q8. Soient a, b deux lettres tel que Σ contiennent a et b . Montrons que $\text{Pal}(\Sigma)$ n'est pas rationnel.

Procédons par l'absurde : si $\text{Pal}(\Sigma)$ est rationnel, alors le langage $L = \text{Pal}(\Sigma) \cap a^*ba^*$ est également rationnel par intersection. Alors par le théorème de Kleene, il existe un automate qui reconnaît L . Supposons qu'il a N états. Considérons le mot $u = a^Nba^N$.

On peut appliquer le lemme de l'étoile et obtenir ce qui suit : il existe une décomposition $u = xyz$ avec y non vide telle que $L(xy^*z) \subseteq L$. On distingue plusieurs cas :

- Si y ne contient que des a , alors y est intégralement contenu dans un des deux facteurs a^N de u . Comme y est non vide, le mot xz est alors de la forme a^pba^q avec $p \neq 0$, donc ce n'est pas un palindrome. Alors $xz \notin L$, ce qui contredit le résultat du lemme de l'étoile.
- Si y contient au moins un b , alors le mot xy^2z n'appartient pas à L , ce qui donne également une contradiction.

Dans tous les cas, on a alors une contradiction. Donc le langage $\text{Pal}(\Sigma)$ n'est pas rationnel.

Q9. Le langage $L_{q,q'}$ est reconnaissable par l'automate $A_{q,q'} = (Q, q, q', T)$ par construction. On peut alors exprimer L_A comme l'union de ces langages pour $q \in I$ et $q' \in F$:

$$L_A = \bigcup_{q \in I, q' \in F} L_{q,q'}$$

Q10. Procédons par double inclusion :

- $\boxed{\subseteq}$ On note $L = u \in \text{Pal}(\Sigma) \cap (\Sigma^2)^*$. Par construction, tous ses mots sont des palindromes contenant un nombre pair de lettres ; ils sont donc tous de longueur paire. On peut alors montrer par récurrence sur la longueur n de ces mots qu'ils sont tous de la forme $u\tilde{u}$:
 - Si $n = 0$, le seul mot possible est le mot vide, et on a bien $\varepsilon = \varepsilon\tilde{\varepsilon}$.
 - Si $w \in L$ est un mot de longueur $n + 2$, remarquons que c'est en particulier un palindrome, donc il s'écrit $w = ava$, où $v \in L$ est un mot de longueur n . Par hypothèse de récurrence, v est de la forme $u\tilde{u}$ pour $u \in \Sigma^*$. Alors on a $w = (au)(\tilde{u}a) = (au)(\tilde{a}u)$.
- $\boxed{\supseteq}$ Si $v = u\tilde{u}$ pour un certain mot u , alors :
 - v est de longueur paire car $|v| = |u| + |\tilde{u}| = 2|u|$. Donc $v \in (\Sigma^2)^*$.
 - $\tilde{v} = \tilde{u\tilde{u}} = u_1u_2 \dots u_n\tilde{u}_n \dots u_2u_1 = u_1u_2 \dots u_nu_n \dots u_2u_1 = u\tilde{u} = v$. Donc $v \in \text{Pal}(\Sigma)$.

Finalement, $v \in L$.

Q11.

- Pour $L = L(a^*b)$, tout $v \in D(L)$ s'écrit $u\tilde{u}$ avec u de la forme $a^n b$ avec $n \in \mathbb{N}$. Donc v est de la forme $a^n b b a^n$ avec $n \in \mathbb{N}$. On en déduit que $D(a^*b) = \{a^n b b a^n \mid n \in \mathbb{N}\} = \text{Pal}(\Sigma) \cap a^*b^2a^*$.
- Si $u \in R(a^*b^*a^*)$, alors $u\tilde{u}$ doit être de la forme $a^p b^q a^r$ avec $p, q, r \in \mathbb{N}$. Comme $u\tilde{u}$ est un palindrome (d'après la question 10), on doit avoir $p = r$. De plus, il doit être de longueur paire (d'après la question 10), donc q est de la forme $2k$. Finalement, $u = a^p b^k$, et on en déduit $R(L) = L(a^*b^*)$.

Q12.

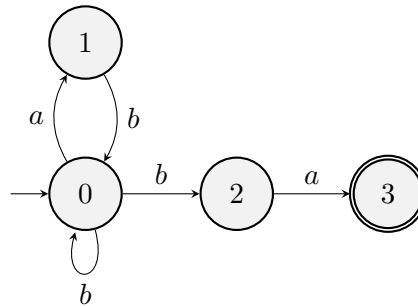
- Le langage $D(L)$ n'est pas reconnaissable par un automate dans le cas général : on a montré à la question précédente que si $L = a^*b$, alors $D(a^*b) = \text{Pal}(\Sigma) \cap a^*b^2a^*$, et on peut montrer à l'aide du lemme de l'étoile et d'un raisonnement par l'absurde (similaire à celui de la question 8) que ce langage n'est pas régulier ; donc d'après le théorème de Kleene, il n'est pas reconnaissable.

- Soit $u \in \Sigma^*$. On a :
 - $u \in R(L)$
 - ssi $u\tilde{u} \in L$
 - ssi $u\tilde{u}$ est accepté par A
 - ssi il existe un chemin de la forme $q_1 \xrightarrow{u}_* q_2 \xrightarrow{\tilde{u}}_* q_3$ avec $(q_1, q_2, q_3) \in I \times Q \times F$ Donc
 - ssi il existe $(q_1, q_2, q_3) \in I \times Q \times F$ tels que $u \in L_{q_1, q_2}$ et $\tilde{u} \in L_{q_2, q_3}$
 - ssi il existe $(q_1, q_2, q_3) \in I \times Q \times F$ tels que $u \in L_{q_1, q_2} \cap \tilde{L}_{q_2, q_3}$
 - ssi $u \in \bigcap_{(q_1, q_2, q_3) \in I \times Q \times F} L_{q_1, q_2} \cap \tilde{L}_{q_2, q_3}$

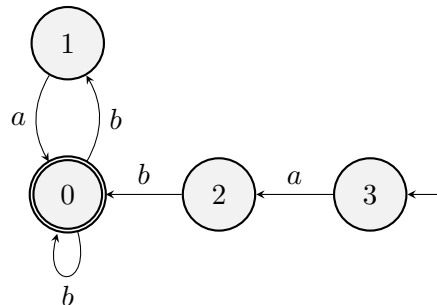
d'après ce qui précède, $R(L) = \bigcap_{(q_1, q_2, q_3) \in I \times Q \times F} L_{q_1, q_2} \cap \tilde{L}_{q_2, q_3}$.

Or, chacun des langages $L_{q, q'}$ est reconnaissable par automate d'après la question 9. De plus, les langages reconnaissable par automate forment une classe stable par intersection finie, et par miroir d'après la question 3. Donc $R(L)$ est reconnaissable par un automate.

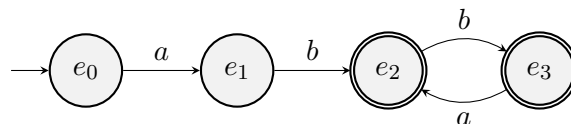
Q13. Voici un automate A_2 qui convient



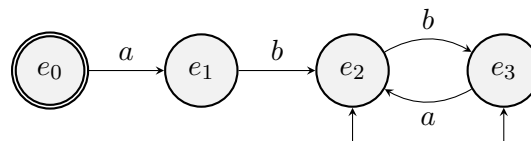
Q14. Donnons d'abord l'automate miroir \tilde{A}_2



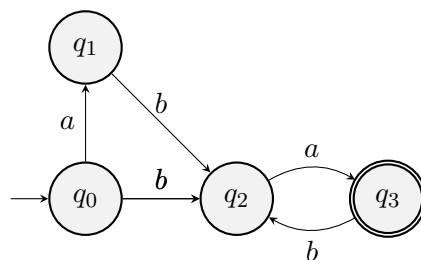
Et voici son déterminisé A_3



Q15. Donnons d'abord l'automate miroir \tilde{A}_3



Et voici son déterminisé A_4



Q16. Etant donné que l'opération de déterminisation conserve le langage reconnu, on a les égalités suivantes : $L_{A_4} = L_{\tilde{A}_3} = \tilde{L}_{A_3} = \tilde{L}_{\tilde{A}_2} = \tilde{\tilde{L}}_{A_2} = L_{A_2}$

Q17. Voici la fonction demandée

```

1 let rec supprimer (l : 'a list) : 'a list =
2     (* fonction auxiliaire supprimant de la liste l'élément
3     donn" en argument *)
4     let rec supprimer_un l x =
5         match l with
6         | [] -> []
7         | a::t -> let l2 = supprimer_un t x in
8                     if a = x then l2 else a::l2
9     in
10    match l with
11    | [] -> []
12    | a::t -> let l2 = supprimer_un t a in a::(supprimer t)
13    (* on supprime toute occurrence de a dans t, pour ne
14    garder que celle-ci *)
  
```

Q18. La fonction auxiliaire `supprimer_un` parcourt linéairement la liste `l` qu'elle reçoit en argument. Sa complexité est donc $O(m)$ où m est la longueur de la liste donnée en entrée.

Si on note n la longueur de la liste en entrée de `supprimer`, le pire des cas est celui où tous les éléments de la liste sont déjà distincts. Dans ce cas, la fonction effectue n appels récursifs, et à chacun d'eux, appelle `supprimer_un` sur une liste de longueur au plus n . Chaque appel récursif a donc une complexité en $O(n)$; comme on en fait au plus n , la complexité totale est en $O(n^2)$.

Q19. On remarque ce qui suit : si X est un ensemble d'états, alors `numero(X)` peut être vu comme un nombre en binaire tel que le i -ème chiffre en partant de la droite (en commençant à $i = 0$) est un 1 si et seulement si $i \in X$.

Il suffit donc, pour répondre à la question posée, de regarder la valeur du q -ème bit en partant de la droite.

```

1 let est_dans (q : int) (k : int) : bool =
2     ((k/pow.(q)) mod 2) = 1
  
```

Q20. Voici la fonction demandée :

```
1 let numero (l : int list) : int =
2     (* fonction auxiliaire calculant k a partir d'une liste d'
3     elements distincts *)
4     let rec aux (li : int list) (res : int) : int =
5         match li with
6         | [] -> res
7         | a::t -> aux t (res+pow.(a))
8     in
9     aux (supprime l) 0
```

On remarquera en particulier que la liste vide, représentant l'ensemble vide, est encodée par 0.

Q21. Voici la fonction demandée :

```
1 let rec intersekte (l : int list) (k : int) : bool =
2     match l with
3     | [] -> false
4     | a::t -> if est_dans a k then true else intersekte t k
```

On peut également utiliser `List.exists` :

```
1 let rec intersekte (l : int list) (k : int) : bool =
2     List.exists (fun q -> est_dans q k) l
```

Q22. On va utiliser la stratégie suivante : on parcourt `T` dans une fonction auxiliaire pour créer deux listes `la` et `lb`, représentant respectivement les ensembles $\delta(X,a)$ et $\delta(X,b)$, puis utiliser les fonctions précédentes.

```
1 let rec etat_suivant (k : int) (delta : (int * char * int) list )
2     : int * int =
3     (* création des listes la et lb *)
4     let rec creer_listes del la lb =
5         match del with
6         | [] -> (la,lb)
7         | (q1,lettre,q2)::t -> let b = est_dans q1 k in
8             let b' = (lettre = 'a') in
9             if b && b' then creer_listes t (
10                 q1::la) lb
11             else if b then creer_listes t la
12                 (q1::lb)
13             else creer_listes t la lb
14         in
15     let (la,lb) = creer_listes delta [] [] in
16     (numero la, numero lb)
```

Q23. Voici la fonction demandée

```
1 let rec cherche (k : int) (l : (int * int) list ) : int =
2     match l with
```

```

3 | [] -> -1
4 | (q,x)::t -> if q = k then x else cherche k t

```

Q24. Pour construire l'automate déterminisé, on va essentiellement respecter les points suivants :

- on conserve dans un accumulateur (à la manière d'une pile) les nouveaux états du déterminisé qu'on génère et qu'il faut encore traiter.
- on conserve dans une liste les renommages à effectuer, dans l'ordre où on rencontre les états. Cela permet également de savoir quels états ont déjà été rencontrés.
- on conserve trace du prochain indice disponible pour connaître l'entier par lequel on renommatera chaque état.
- si l'accumulateur est vide, on a fini. Sinon, on prend le premier état de la liste et on génère les états suivants.

```

1 let rec determinise (a : automate) : automate =
2     let n = pow.(a.nb) in
3
4     (* acc : etats a traiter *)
5     (* i : prochain indice *)
6     (* sigma : liste de renommage *)
7     (* res : transitions *)
8     (* resf : etats finaux *)
9     let aux_etape acc i sigma res resf =
10        match acc with
11        | [] -> (i,sigma,res,resf)
12        | q::t -> if cherche q sigma = -1 then
13            begin
14
15                (* on ajoute cet etat *)
16                let new_sigma = (q,i)::sigma in
17
18                (* on calcule les etats suivants*)
19                let (qa,qb) = etat_suivant q (a.
20                    delta) in
21
22                (* on cherche s'ils ont deja ete
23                    rencontres *)
24                let b_qa = (cherche qa sigma = -1)
25                    in
26                let b_qb = (cherche qb sigma = -1)
27                    in
28
29                (* on ajoute éventuellement ces
30                    transitions *)
31                let new_res =
32                    if b_qa then res else (q,'
33                        a',qa)::res in

```

```

28         let newer_res =
29             if b_qa then new_res else
30                 (q,'b',qb)::new_res in
31
32         (* on determine si ces etats sont
33            finaux *)
34         let b1 = intersecte qa a.final in
35         let b2 = intersecte qb a.final in
36         let newer_resf =
37             if (not b_qa) && b1 then
38                 qa::resf
39             else
40                 resf
41         in let newer_resf =
42             if (not b_qb) && b2 then
43                 qa::newer_resf
44             else
45                 newer_resf
46         in
47
48         (* on continue*)
49         aux_etape t (i+1) new_sigma
50             newer_res newer_resf
51         end
52     else
53         aux_etape acc t i res
54
55     (* on construit enfin les champs du nouvel automate *)
56     let init = numero (a.init) in
57     let (i,sigma,trans,f) = aux_etape [init] 0 [] [] [] in
58
59     let f_det = List.map (fun x -> cherche x sigma) f in
60     let trans_det = List.map (fun x -> let (q1,a,q2) = x in (
61         cherche q1, a, cherche q2)) trans in
62
63     (* et on renomme les états *)
64     { nb = i ; init = [0] ; final = f_det ; trans = trans_det
65       }

```

Q25.

- Commençons par les dernières étapes : chaque "renommage" à l'aide de la fonction `cherche` a une complexité en $O(N)$, car on cherche le nouveau numéro d'un état du déterminisé dans une liste.
On fait au plus N renommages dans la liste des états finaux, puis au plus $4N$ dans la liste des transitions (car chaque état a au plus deux transitions sortantes, et que chacune est encodée par un triplet contenant deux entiers).
Au total, la complexité de ces renommages est en $O(5N \times N) = O(N^2)$.
- Cherchons maintenant la complexité de la partie récursive :

- Si on tombe sur un état déjà rencontré, on ne fait que refaire un appel récursif sans autre opération. Le coût de ces opérations, sans l'appel récursif, est donc en $O(1)$, et ceci peut se produire au plus $O(N)$ fois, puisque l'automate a au plus $2N$ transitions.
- Si on tombe sur un état jamais rencontré, ce qui arrive exactement N fois, on procède aux opérations suivantes :
 - * Calculer les états suivants. Ceci se fait à l'aide de la fonction `etat_suivant`, dont la complexité est en $O(|T|)$ (nombre de transitions de l'automate d'origine), lui même majoré par $O(n^2)$.
 - * Chercher si les états suivants ont déjà été rencontrés. Ceci se fait en $O(N)$.
 - * Chercher éventuellement si les états suivants sont acceptants. Ceci se fait à l'aide de la fonction `intersecte`. Celle-ci parcourt la liste des états finaux de l'automate d'origine, et fait des opérations en temps constant pour chacun. La complexité est donc en $O(n)$.

La complexité totale de ces opérations est donc en $O(N + n^2)$. Puis on exécute les appels récursifs.

La complexité totale est alors $O(N + N^2 + N \times (N + n^2)) = O(N^2 + Nn^2)$.

Propriété intéressante : avant de traiter les questions suivantes, on peut montrer cette propriété qui servira dans plusieurs questions : avec les hypothèses de l'énoncé, pour tout mot $u \in \Sigma^*$, il existe au plus un unique état $q \in Q$ tel que la lecture de u depuis l'état q amène à l'état f .

Cela peut se montrer par récurrence sur la longueur de u :

- Si $u = \varepsilon$, c'est évident.
- Sinon, u s'écrit $u = av$. Par hypothèse de récurrence, il existe au plus un $q' \in Q$ tel que la lecture de v depuis q' amène à l'état f . On utilise maintenant le fait que l'automate miroir est accessible et déterministe : la lecture de la lettre a depuis q' amène vers au plus un état q . Par miroir, il existe alors au plus un unique état q tel que la lecture de u depuis q amène dans l'état f .

Q26. L'automate miroir \tilde{A} est accessible par hypothèse, chacun de ses états sont donc accessibles. En particulier, cela implique que tous les états de A sont co-accessibles.

Si $q \in \delta^*(\{I\}, u)$, cela signifie que l'état q est accessible dans l'automate A en lisant le mot u depuis un état initial de A ; de plus, q est co-accessible d'après ce qui précède. Alors il existe un mot w qui est l'étiquette d'un chemin depuis q jusqu'à un état final de A .

Cet assemblage nous permet de construire un chemin acceptant dans A , d'étiquette uw . Il existe donc bien un $w \in \Sigma^*$ tel que $uw \in L$.

Q27. On peut procéder par double inclusion ; et comme u et v jouent des rôles symétriques dans l'énoncé, on peut ne montrer qu'une seule inclusion, car l'autre se traitera de la même manière. Supposons donc que $u^{-1}L = v^{-1}L$ et montrons que $\delta^*(\{I\}, u) \subset \delta^*(\{I\}, v)$.

Soit $q \in \delta^*(\{I\}, u)$. D'après la question précédente, il existe w tel que $uw \in L$. Donc $w \in u^{-1}L = v^{-1}L$; alors $vw \in L$. Il existe donc un état dans $\delta^*(\{I\}, v)$ depuis lequel on peut lire w pour arriver dans f . Or, d'après la propriété montrée précédemment, cet état est nécessairement q . Donc $q \in \delta^*(\{I\}, v)$.

Q28.

- Comme l'opération de détermination conserve les langages reconnus, l'automate $B = (\tilde{A})_{det}$ reconnaît le même langage que \tilde{A} , c'est-à-dire \tilde{L} . Alors $(\tilde{B})_{det}$ reconnaît le même langage que \tilde{B} , c'est-à-dire $\tilde{L} = L$.
- Comme B est l'automate déterminisé accessible de \tilde{A} , il est en particulier déterministe et accessible, et c'est l'automate miroir de \tilde{B} . Donc l'automate \tilde{B} vérifie les hypothèses formulées au début de la partie *I.D*, qu'on a utilisées pour montrer la propriété (*). Donc $(\tilde{B})_{det}$ vérifie la propriété (*);.

Q29. Voici la fonction demandée :

```
1 let minimal (a : automate) : automate = determinise (transpose (
  determinise (transpose a)))
```

II Expression rationnelle associée à un automate

Q30. Voici la fonction demandée :

```
1 let rec lettre (e : exprat) : int =
2     match e with
3     | Vide -> 0
4     | Epsilon -> 0
5     | Lettre _ -> 1
6     | Union(e1,e2) -> (lettre e1) + (lettre e2)
7     | Concat(e1,e2) -> (lettre e1) + (lettre e2)
8     | Etoile(e1) -> lettre e1
```

Q31. Voici la fonction demandée :

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

Q32. Voici la fonction demandée :

```
1 let se (e : exprat) : exprat =
2     match e with
3     | Etoile(Epsilon) -> Epsilon
4     | Etoile(Etoile(e1)) -> Etoile(e1)
5     | _ -> e
```

Q33. En appliquant une fois la simplification de la concaténation sur l'arbre le plus profond de E_n , on la transforme en E_{n-1} . Par récurrence, on montre qu'en l'appliquant n fois, on la transforme en $E_0 = a + \emptyset$. Enfin, il suffit d'appliquer une fois la simplification de l'union pour obtenir a .

Il faut alors appliquer $n + 1$ simplifications de règles.

Q34. Voici la fonction demandée :

```

1 let rec simplifie (e : exprat) : exprat =
2     match e with
3     | Union(e1,e2) -> let e1' = simplifie e1 and e2' =
4         simplifie e2 in
5         su (Union(e1',e2'))
6     | Concat(e1,e2) -> let e1' = simplifie e1 and e2' =
7         simplifie e2 in
8         sc (Concat(e1',e2'))
9     | Etoile(e1) -> let e1' = simplifie e1 in
10        se (Etoile(e1'))
11    | _ -> e

```

Q35. Voici la fonction demandée.

```

1 let rec somme (a : mat) (b : mat) : mat =
2     (* on vérifie que les deux matrices ont le même nombre de
3     lignes *)
4     let na = Array.length a and nb = Array.length b in
5     if (na <> nb) then failwith "Erreur : nombre de lignes
6     incompatible dans somme"
7     else
8     begin
9         let c = Array.make_matrix (na) (Array.length (a
10         .(0))) Vide in
11         for i = 0 to na-1 do
12             (* on vérifie que les i-ème lignes des
13             deux matrices ont le même nombre de
14             cases *)
15             let pa = Array.length a.(i) and pb = Array
16             .length b.(i) in
17             if (pa <> pb) then failwith "Erreur :
18             nombre de colonnes incompatible dans
19             somme"
20             else
21             begin
22                 for j = 0 to pa-1 do
23                     c.(i).(j) <- Union(a.(i).(
24                     j),b.(i).(j))
25                 done
26             end
27         done;
28         c

```

end

Les opérations qui majorent la complexité sont l'exécution des deux boucles. Pour deux matrices de taille $(n \times p)$, la boucle externe effectue n itérations, et la boucle interne en effectue p . Le contenu de la boucle interne s'effectue en temps constant. Le nombre d'opérations est donc proportionnel à np , et la complexité est alors en $O(np)$.

Q36. Voici la fonction demandée.

```

1 let rec somme (a : mat) (b : mat) : mat =
2     (* on vérifie que les deux matrices ont le même nombre de
3     lignes *)
4     let n = Array.length a and p = Array.length b and q =
5     Array.length b.(0) in
6     let c = Array.make_matrix n q Vide in
7     for i = 0 to n-1 do
8         for j = 0 to q-1 do
9             for k = 0 to p-1 do
10                c.(i).(j) <- Union(c.(i).(j),
11                Concat(a.(i).(k), b.(k).(j)))
12            done;
13        done;
14    done;
15    c

```

La complexité de cette fonction est proportionnelle au nombre d'itérations de ses trois boucles. Avec les notations de l'énoncé, elle est en $O(npq)$.

Q37. Avec ces notations, on a les expressions suivantes pour chaque langage :

- $L_{0,0} : (a + bd^*c)^*$
- $L_{0,1} : a^*b(d + ca^*b)^*$
- $L_{1,0} : d^*c(a + bd^*c)^*$
- $L_{1,1} : (d + ca^*b)^*$

Q38. Comptons les opérations importantes dans le calcul des 4 blocs précédents.

- Les matrices D^* et $(D + Ca^*B)^*$ sont des étoiles de matrices de taille $(n - 1 \times n - 1)$. Comme elles apparaissent à plusieurs reprises, il suffit de les calculer une fois chacune, ce qui se fait en $C(n - 1)$ pour chacune d'elles une fois que le calcul de la matrice interne est fini.
- Le calcul de a^* et de $(a + BD^*C)^*$ se fait en $C(1)$ une fois les matrices internes calculées, car il s'agit de matrices de taille (1×1) . Ce coût sera vraisemblablement négligé devant les coûts évoqués après.
- Le calcul des unions dans A' et dans C' est en $O(1)$, car il est appliqué à des matrices de taille 1×1 .

- Il reste le calcul des différents produits. Dans A' , le produit BD^*C implique des matrices de tailles respectives $(1 \times n - 1)$, $(n - 1 \times n - 1)$, $(n - 1 \times 1)$. En calculant les deux produits successivement, on a d'abord un produit en $O((n - 1)^2)$, puis un produit en $O(n - 1)$. La somme des coûts de ces deux opérations est alors majorée asymptotiquement par $O(n^2)$.

On peut appliquer le même raisonnement pour les autres produits dans les autres blocs, et tous leurs coûts sont majorés par $O(n^2)$.

Au total, cela montre que le calcul de M^* nécessite deux calculs d'étoiles de complexité $C(n - 1)$, et d'autres opérations dont la complexité est majorée asymptotiquement par $O(n^2)$, d'où la formule de récurrence $C(n) = 2C(n - 1) + O(n^2)$.

$$\begin{aligned} C(n) &= 2C(n - 1) + O(n^2) \\ &= 4C(n - 2) + 2O(n^2) + O(n^2) \end{aligned}$$

Itérons cette relation de récurrence :

$$\begin{aligned} &\vdots \\ &= 2^{n-1}C(1) + (2^{n-2} - 1)O(n^2) \\ &= O(2^n n^2) \end{aligned}$$

Q39. On a désormais 4 blocs de taille $\frac{n}{2}$, et toutes les matrices impliquées dans le calcul de A', B', C', D' sont de taille $(\frac{n}{2} \times \frac{n}{2})$. Comptons alors les opérations importantes :

- Il y a 4 calculs d'étoile : D^* , A^* , $(A + BD^*C)^*$ et $(D + CA^*B)^*$. Ceux-ci contribuent pour une complexité de $4C(\frac{n}{2})$.
- Il y a 12 produits, chacun impliquant deux matrices de taille $(\frac{n}{2} \times \frac{n}{2})$. Chacun compte alors pour une complexité de $O(\frac{n^3}{8}) = O(n^3)$.
- Il y a 4 sommes, chacune compte pour $O(\frac{n^2}{4})$.

Au total, la complexité de $C(n)$ s'exprime alors comme suit : $C(n) = 4C(\frac{n}{2}) + O(n^3)$.

$$\begin{aligned} C(n) &= 4C(\frac{n}{2}) + O(n^3) \\ &= 16C(\frac{n}{4}) + 4O(n^3) + O(n^3) \\ &= 64C(\frac{n}{8}) + 16O(n^3) + 4O(n^3) + O(n^3) \end{aligned}$$

Itérons cette relation de récurrence :

$$\begin{aligned} &\vdots \\ &= 4^{\log_2(n)}C(1) + (4^{\log_2(n)} - 1)O(n^3) \\ &= O(n^5) \end{aligned}$$

Q40. Si une matrice M est de taille quelconque, on peut utiliser la procédure suivant : si n est pair, on divise en des blocs de taille identique ; si n est impair, on pose $n = 2p + 1$ et on divise en des blocs de taille p et $p + 1$. En appliquant un raisonnement similaire au raisonnement précédent, on peut se ramener à une complexité en $O(n^5)$ à nouveau.

```

1 let rec etoile (m : mat) : mat =
2     let n = Array.length m in
3     let p = n/2 in
4     let q =
5         if n mod 2 = 0 then p else p+1
6     in
7
8     (* a ce stade, on a toujours p+q = n *)
9     let (a,b,c,d) = decouper m p in
10    let de = etoile d in
11    let ae = etoile a in
12    let ae_b = produit ae b in
13    let de_c = produit de c in
14    let bdec = produit b de_c in
15    let caeb = produit c ae_b in
16    let a0 = somme a bdec in
17    let d0 = somme d caeb in
18
19    let a' = etoile a0 in
20    let d' = etoile d0 in
21    let b' = produit ae_b d0 in
22    let c' = produit de_c a0 in
23
24    recolle a' b' c' d'

```

Q42. Posons $X = (x_0 \ \dots \ x_{n-1})$ avec pour tout i , $x_i = \varepsilon$ si $i \in I$, et $x_i = \emptyset$ sinon.

On définit de même $Y = \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix}$ avec pour tout i , $y_i = \varepsilon$ si $i \in F$, et $y_i = \emptyset$ sinon.

On note $[M_A^*]_{i,j} = e_{i,j}$ les expressions rationnelles qui servent de coefficients à la matrice M_A^* . On peut alors calculer XM_A^* , puis XM_A^*Y .

XM_A^* est une matrice de taille $(1 \times n)$, de terme général $t_j = \sum_{i=0}^{n-1} x_i e_{i,j} \equiv \sum_{i \in I} e_{i,j}$. Et donc XM_A^*Y est une matrice de taille (1×1) , dont le seul coefficient est $\sum_{j=0}^{n-1} t_j y_j \equiv \sum_{j \in F} t_j \equiv \sum_{i \in I} \sum_{j \in F} e_{i,j}$.

Ainsi, on a $L([XM_A^*Y]_{0,0}) = L(\sum_{i \in I} \sum_{j \in F} e_{i,j}) = \bigcup_{i \in I, j \in F} L_{i,j} = L_A$.

Q43. Voici la fonction demandée :

```

1 let langage (a : automate) : exprat =
2     (* on réserve la mémoire nécessaire *)
3     let n = a.nb in
4     let x = Array.make_matrix 1 n Vide in
5     let y = Array.make_matrix n 1 Vide in

```

```

6      let m = Array.make_matrix n n Vide in
7
8      (* on initialise chaque case *)
9      let rec init_x li =
10         match li with
11         | [] -> ()
12         | i::l -> x.(0).(i) <- Epsilon ; init_x l
13
14     in
15     let rec init_y li =
16         match li with
17         | [] -> ()
18         | i::l -> y.(i).(0) <- Epsilon ; init_y l
19
20     in
21     let rec init_m li =
22         match li with
23         | [] -> ()
24         | (q0,a,q1)::l -> m.(q0).(q1) <- Union(m.(q0).(q1)
25             , Lettre a) ; init_m l
26
27     in
28     init_x (a.init) ;
29     init_y (a.final) ;
30     init_m (a.trans) ;
31
32     (* on calcule enfin le produit matriciel *)
33     let me = etoile m in
34     let xme = produit x me in
35     let xmey = produit xme y in
36
37     xmey.(0).(0)

```

Si on appelle n le nombre d'états de l'automate d'entrée, et qu'on suppose fixée la taille de Σ , alors :

- la réservation en mémoire des trois tableaux coûte au plus $O(n^2)$.
- l'initialisation des trois tableaux coûte au plus $O(|I| + |F| + |T|) = O(n^2)$
- le calcul de l'étoile coûte au plus $O(n^5)$.
- Le calcul des produits coûte au plus $O(n^2)$.

La complexité est alors majorée par le calcul de l'étoile : $O(n^5)$.

III Automate des dérivées d'Antimirov

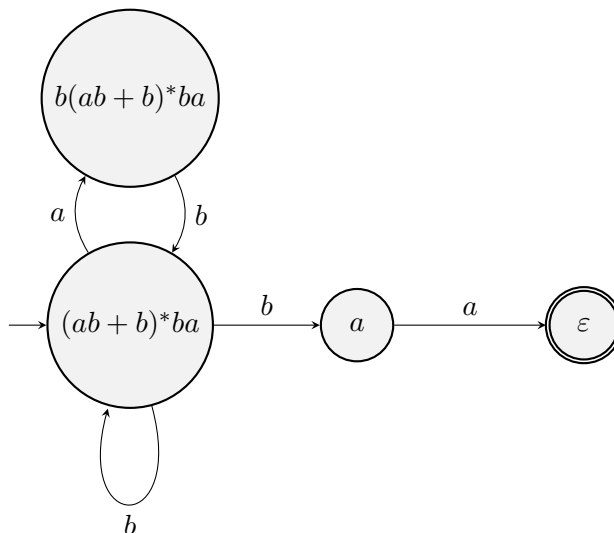
$$\begin{aligned}
 \partial_a(E) = \partial_a((ab + b)^*ba) &= \partial_a((ab + b)^*) \cdot \{ba\} \cup \partial_a(ba) \\
 &= \partial_a(ab + b) \cdot \{(ab + b)^*\} \cdot \{ba\} \cup \partial_a(b) \cdot \{a\} \\
 &= \partial_a(ab + b) \cdot \{(ab + b)^*ba\} \cup \emptyset \\
 &= (\partial_a(ab) \cup \partial_a(b)) \cdot \{(ab + b)^*ba\} \\
 &= (\partial_a(a) \cdot \{b\}) \cdot \{(ab + b)^*ba\} \\
 &= \{b(ab + b)^*ba\}
 \end{aligned}$$

Q44.

$$\begin{aligned}
 \partial_b(E) = \partial_b((ab + b)^*ba) &= \partial_b((ab + b)^*) \cdot \{ba\} \cup \partial_b(ba) \\
 &= \partial_b((ab + b)^*) \cdot \{ba\} \cup \partial_b(b) \cdot \{a\} \\
 &= \partial_b(ab + b) \cdot \{(ab + b)^*\} \cdot \{ba\} \cup \{a\} \\
 &= (\partial_b(ab) \cup \partial_b(b)) \cdot \{(ab + b)^*ba\} \cup \{a\} \\
 &= (\emptyset \cup \{\varepsilon\}) \cdot \{(ab + b)^*ba\} \cup \{a\} \\
 &= \{(ab + b)^*ba, a\}
 \end{aligned}$$

Q45. L'ensemble d'états obtenus est l'ensemble d'expressions rationnelles suivant : $Q = \{E, b(ab + b)^*ba, a, \varepsilon\}$.

L'automate obtenu est alors précisément l'automate A_2 de la question 13 :



Q46. On peut procéder par équivalence directement :

$$\begin{aligned}
 w \in v^{-1}u^{-1}L &\text{ ssi } vw \in u^{-1}L \\
 &\text{ ssi } (uv)w \in L \\
 &\text{ ssi } w \in (uv)^{-1}L
 \end{aligned}$$

Q47. On peut procéder par récurrence sur la longueur de w :

- Si $w = \varepsilon$, on a $L(\partial_\varepsilon(S)) = L(\bigcup_{E \in S} \partial_\varepsilon(E)) = L(\bigcup_{E \in S} \{E\}) = L(S)$
- Si $w = ua$, on a :

$$\begin{aligned}
L(\partial_{ua}(S)) &= L\left(\bigcup_{E \in S} \partial_{ua}(E)\right) \\
&= L\left(\bigcup_{E \in S} \partial_a(\partial_u(E))\right) \\
&= \bigcup_{E \in S} L(\partial_a(\partial_u(E))) \\
&= \bigcup_{E \in S} a^{-1}L(\partial_u(E)) \\
&= \bigcup_{E \in S} a^{-1}u^{-1}L(E) \quad (\text{hypothèse de récurrence}) \\
&= \bigcup_{E \in S} (ua)^{-1}L(E) \quad (\text{question 46}) \\
&= (ua)^{-1}\left(\bigcup_{E \in S} L(E)\right) \quad (\text{se vérifie facilement}) \\
&= w^{-1}L(S)
\end{aligned}$$

Q48. On peut procéder par récurrence sur la longueur de w :

- Si $w = \varepsilon$, on a $\partial_\varepsilon(E) = \{E\}$, qui est bien l'ensemble des états accessible depuis E en lisant ε .
- Si $w = ua$, on a $\partial_{ua}(E) = \partial_a(\partial_u(E))$. Par hypothèse de récurrence, $E' = \partial_u(E)$ est l'ensemble des états accessibles depuis E en lisant u .

On remarque alors que :

$$\begin{aligned}
e \in \partial_{ua}(E) &\text{ ssi } e \in \partial_a(\partial_u(E)) \\
&\text{ ssi il existe } s \in \partial_u(E) \text{ tel que } e \in \partial_a(s) \\
&\text{ ssi la transition } (s, a, e) \text{ existe et } s \text{ est accessible depuis } E \text{ en lisant } u \\
&\text{ ssi } e \text{ est accessible depuis } E \text{ en lisant } ua = w.
\end{aligned}$$

Q49. On peut procéder par équivalence :

$$\begin{aligned}
w \in L(E) &\text{ ssi } w\varepsilon \in L(E) && (\text{trivial}) \\
&\text{ ssi } \varepsilon \in w^{-1}L(E) && (\text{définition}) \\
&\text{ ssi } \varepsilon \in L(\partial_w(E)) && (\text{question 47}) \\
&\text{ ssi il existe } E' \in \partial_w(E) \text{ telle que } \varepsilon \in L(E') \\
&\text{ ssi il existe } E' \in \partial_w(E) \text{ telle que } E' \in F && (\text{définition}) \\
&\text{ ssi il existe } E' \in F \text{ accessible depuis } E \text{ en lisant } w && (\text{question 48}) \\
&\text{ ssi } w \text{ est accepté par l'automate d'Antimirov} && (\text{définition})
\end{aligned}$$

Q50. On procède par induction sur E .

- Si $E = \emptyset$, on a $Q(E) = \emptyset$ et donc $|Q(E)| = 0 \leq \|E\| = 0$
- Si $E = \varepsilon$, on a $Q(E) = \emptyset$ et donc $|Q(E)| = 0 \leq \|E\| = 0$
- Si $E = a$, on a $Q(E) = \{\varepsilon\}$ et donc $|Q(E)| = 1 \leq \|E\| = 1$
- Si $E = E_1 + E_2$, on a

$$\begin{aligned}
Q(E) &= \bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_1 + E_2) = \bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} (\partial_w(E_1) \cup \partial_w(E_2)) \\
&= \left(\bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_1) \right) \cup \left(\bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_2) \right) = Q(E_1) \cup Q(E_2)
\end{aligned}$$

$$\text{et donc } |Q(E)| \leq |Q(E_1)| + |Q(E_2)| \leq \|E_1\| + \|E_2\| = \|E\|$$

- Si $E = E_1E_2$, on a

$$\begin{aligned}
Q(E) &= \bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_1E_2) \subset \bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} (\partial_w(E_1) \cdot E_2 \cup \bigcup_{v \in S^+(w)} \partial_v(E_2)) \\
&= \left(\bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_1) \cdot E_2 \right) \cup \left(\bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \bigcup_{v \in S^+(w)} \partial_v(E_2) \right) \\
&= \left(\bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_1) \right) \cdot E_2 \cup \left(\bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_2) \right) \\
&= Q(E_1) \cdot E_2 \cup Q(E_2)
\end{aligned}$$

et donc $|Q(E)| \leq |Q(E_1) \cdot E_2| + |Q(E_2)| = |Q(E_1)| + |Q(E_2)| \leq \|E_1\| + \|E_2\| = \|E\|$

- Si $E = E_1^*$, on a

$$\begin{aligned}
Q(E) &= \bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_1^*) \subset \bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \left(\bigcup_{v \in S^+(w)} \partial_v(E_1) \cdot E_1^* \right) \\
&= \bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} (\partial_w(E_1) \cdot E_1^*) = \left(\bigcup_{w \in \Sigma^* \setminus \{\varepsilon\}} \partial_w(E_1) \right) \cdot E_1^* \\
&= Q(E_1) \cdot E_1^*
\end{aligned}$$

et donc $|Q(E)| \leq |Q(E_1) \cdot E_1^*| = |Q(E_1)| \leq \|E_1\| = \|E\|$

Ainsi, on remarque que pour toute expression rationnelle E , $Q(E)$ est un ensemble fini. Or, $Q(E)$ est précisément l'ensemble des états de l'automate d'Antimirov de l'expression E . Alors cet automate admet bien un nombre fini d'états.