# Option Informatique Mines-Ponts Filière MP 2025

Proposition de corrigé – Nathaniel Carré

⋆⋆⋆

## 1 Alphabets, mots et automates

**Question 1** Comme il n'y a que deux lettres dans l'alphabet, il suffit d'envisager les quatre cas possibles.

```
let cmp_letter x1 x2 = match x1, x2 with
    | A, B -> -1
    | B, A -> 1
    | _ -> 0
```

**Question 2** On compare récursivement lettre à lettre, jusqu'à ce qu'un des deux mots soit vide.

```
let rec cmp_word w1 w2 = match w1, w2 with
    | [], [] -> 0
    | _, [] -> -1
    | [], _ -> 1
    | x1 :: u1, x2 :: u2 ->
        let c = cmp_letter x1 x2 in
        if c = 0 then cmp_word u1 u2
        else c
```

**Question 3** Les arbres binaires de recherche permettent d'implémenter des dictionnaires immuables. On peut pour cela utiliser un type classique d'arbre comme :

```
type 'a dict = Vide | N of 'a dict * word * 'a * 'a dict
```

où un dictionnaire est soit un arbre vide, soit un nœud formé d'une clé (un mot), une valeur et de deux enfants qui sont des dictionnaires.

Il faut que les clés soient distinctes pour qu'on puisse retrouver la bonne valeur. Par ailleurs, la structure d'arbre binaire de recherche garantit que les clés dans l'ordre du parcours en profondeur infixe de l'arbre soient croissantes.

**Question 4** Il s'agit juste d'une question de syntaxe.

```
let delta_a =
    WordMap.add [] [A] (WordMap.add [A] [] WordMap.empty)

let delta_b =
    WordMap.add [] [] (WordMap.add [A] [A] WordMap.empty)
```

```
let un_a =
    WordMap.add [] false (WordMap.add [A] true WordMap.empty)
```

**Question 5** Le langage reconnu est l'ensemble des mots sur l'alphabet $\{a, b\}$ ayant un nombre impair de $a$. Il correspond à l'expression régulière $(b^*ab^*a)^*b^*ab^*$ par exemple.

**Question 6** Il n'existe pas d'expression régulière ayant cet automate comme automate de Glushkov, car l'automate de Glushkov est toujours un automate standard (pas de transition vers l'état initial), ce qui n'est pas le cas ici.

**Question 7** En demandant de compléter le code, l'énoncé suggère implicitement d'utiliser la question 4.

```
let automaton_example = {
        initial = [];
        transitions = fun q x
            -> WordMap.find q (if x = A then delta_a else delta_b);
        finals = fun q -> WordMap.find q un_a
    }
```

**Question 8** Un classique de programmation sur les automates. Il suffit d'appliquer la formule d'induction $\delta^*(q, xu) = \delta^*(\delta(q, x), u)$.

```
let rec delta_star a q = function
    | [] -> q
    | x :: u -> delta_star a (a.transitions q x) u
```

**Question 9** On se contente de calculer $\delta^*(q_0, w)$ et de vérifier s'il est dans $F$.

```
let accepte a w =
    a.finals (delta_star a a.initial w)
```

## 2 Relation de séparabilité par rapport à un langage

**Question 10** On applique la définition.

```
let separated_by u1 u2 v =
    Oracle.mem (u1 @ v) <> Oracle.mem (u2 @ v)
```

**Question 11** On montre les trois propriétés :

- réflexivité : pour tout $v \in \Sigma^*$, $\mathbb{1}_\Lambda(uv) = \mathbb{1}_\Lambda(uv)$, donc $u \equiv_\Lambda u$ ;
- symétrie : si $u_1 \equiv_\Lambda u_2$, alors pour tout $v \in \Sigma^*$, $\mathbb{1}_\Lambda(u_1 v) = \mathbb{1}_\Lambda(u_2 v)$, donc $\mathbb{1}_\Lambda(u_2 v) = \mathbb{1}_\Lambda(u_1 v)$, soit $u_2 \equiv_\Lambda u_1$ ;

– transitivité : supposons $u_1 \equiv_\Lambda u_2$ et $u_2 \equiv_\Lambda u_3$. Alors pour $v \in \Sigma^*$, $\mathbb{1}_\Lambda(u_1 v) = \mathbb{1}_\Lambda(u_2 v) = \mathbb{1}_\Lambda(u_3 v)$, soit $u_1 \equiv_\Lambda u_3$.

**Question 12** Soit $v \in \Sigma^*$. Notons $q = \delta^*(q_0, u_1) = \delta^*(q_0, u_2)$. Alors $\mathbb{1}_{L_A}(u_1 v) = \mathbb{1}_F(\delta^*(q, v)) = \mathbb{1}_{L_A}(u_2 v)$. On en déduit que $u_1 \equiv_{L_A} u_2$.

**Question 13** Le théorème de Kleene dit que les langages réguliers sont exactement les langages reconnaissables. Ainsi, le langage $L$, supposé régulier, est reconnu par un automate $A = (Q, q_0, \delta, \mathbb{1}_F)$. Par la question précédente, l'automate $A$ étant supposé complet, il y a au plus $|Q|$ classes d'équivalence pour la relation $\equiv_L$, et ce nombre est bien fini.

## 3 Arbre discriminant

**Question 14** Pour obtenir une complexité linéaire en le nombre de feuilles, on ne peut pas se permettre de faire le travail en calculant récursivement les listes des feuilles des enfants et en les concaténant (sinon la complexité pourrait être quadratique).

On peut le faire en purement fonctionnel avec un accumulateur, mais il peut être plus simple à comprendre d'utiliser une référence qu'on complète au fur et à mesure qu'on parcourt les feuilles.

```
let states_of_disctree t =
    let leafs = ref [] in
    let rec fill = function
        | Leaf q -> leafs := q :: !leafs
        | Node (l, _, r) -> fill l; fill r
    in
    fill t;
    !leafs
```

À noter, la liste renvoyée contient les feuilles dans l'ordre de droite à gauche (mais l'ordre n'importe pas).

La complexité est bien linéaire. En effet, il y a un appel récursif par nœud de l'arbre, et les opérations qui ne sont pas des appels récursifs se font en temps constant. La complexité est donc linéaire en la taille de l'arbre. Par ailleurs, l'arbre $T$ étant binaire strict, le nombre de feuille est exactement $\frac{|T|-1}{2}$, ce qui conclut.

**Question 15** On remarque que pour le mot $u = ba$, un mot $uv$ ne peut jamais appartenir à $\mathcal{L}(ab^*a)$, car il ne commence pas par $a$. On en déduit qu'en criblant le mot $u$ à travers $\mathcal{T}_0$, on ira jusqu'à la feuille la plus à droite, donc $\lfloor ba \rfloor = \underline{b}$.

**Question 16** On applique, de manière récursive, la définition du calcul du criblat.

```
let rec sift t u = match t with
    | Leaf q -> q
    | Node (l, v, r) ->
        if Oracle.mem (u @ v) then sift l u
        else sift r u
```

**Question 17** La complexité temporelle est en $\mathcal{O}(|u| \times h \times \mu(|u| + M))$. En effet, on constate qu'à chaque appel récursif, la hauteur de l'arbre donné en argument diminue de 1. Il y a donc au plus $h$ appels récursifs. De plus, chaque appel récursif nécessite de :

- calculer `u @ v`, ce qui se fait en temps $\mathcal{O}(|u|)$ ;
- faire l'appel à l'oracle sur $uv$, ce qui se fait en temps $\mathcal{O}(\mu(|u| + M))$ (en supposant que $\mu$ est une fonction croissante).

**Question 18** Soient $u_1, u_2 \in \Sigma^*$ tels que $\lfloor u_1 \rfloor \neq \lfloor u_2 \rfloor$. Notons $s_0 - s_1 - \cdots - s_k$ et $t_0 - t_1 - \cdots - t_\ell$ les chemins obtenus en criblant $u_1$ et $u_2$ respectivement.

Sachant que $s_0 = t_0$ et que $s_k \neq t_\ell$, il existe un indice $i \geqslant 0$ tel que $s_i = t_i$ et $s_{i+1} \neq t_{i+1}$. Posons $v_i$ le mot au nœud $s_i = t_i$.

Sachant que $s_{i+1} \neq t_{i+1}$, on en déduit que $u_1$ et $u_2$ sont séparés par $v_i$, donc $u_1 \not\equiv_L u_2$.

**Question 19** C'est la contraposée de la question précédente.

**Question 20** D'après la question 13, le nombre de classes d'équivalences d'inséparabilité est fini, majoré par le nombre d'états $n$ d'un automate qui reconnaît $L$.

Soit $T$ un arbre discriminant accessible. Par l'absurde, supposons que $T$ possède strictement plus que $n$ feuilles.

Par le principe des tiroirs, il existe donc deux feuilles distinctes $\underline{w_1}$ et $\underline{w_2}$ telles que $w_1 \equiv_L w_2$. Par la question précédente et par accessibilité :

$$\underline{w_1} = \lfloor w_1 \rfloor = \lfloor w_2 \rfloor = \underline{w_2}$$

ce qui est absurde, car on les a supposées distinctes.

On en déduit que le nombre de feuilles de $T$ est majoré par $n$.

**Question 21** Si $\lfloor u_1 \rfloor = \lfloor u_2 \rfloor$ par rapport à un arbre discriminant démêlant $T$, alors en particulier, le premier test réalisé lors du crible de $u_1$ et $u_2$ à travers $T$ doit être identique, donc $\mathbb{1}_L(u_1\varepsilon) = \mathbb{1}_L(u_2\varepsilon)$.

## 4 Automate tiré d'un arbre discriminant

**Question 22** On applique la définition. À noter, on calcule la liste des états finals avant de définir la fonction indicatrice (pour éviter de recalculer à chaque fois). Notons qu'on aurait pu utiliser un dictionnaire pour créer l'ensemble des états finals ou les transitions, mais la syntaxe aurait été plus lourde.

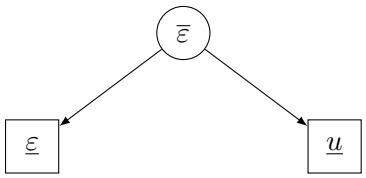
```
let automaton_of_disctree t = {
        initial = [];
        transitions = fun q x -> sift t (q @ [x]);
        finals =
            let final_states =  match t with
                | Leaf _ -> failwith "Pas un crible."
                | Node (l, _, _) -> states_of_disctree l
            in
```

```
            fun q -> List.mem q final_states
    }
```

**Question 23** On commence par faire un appel à `Oracle.mem []` et on distingue deux cas :

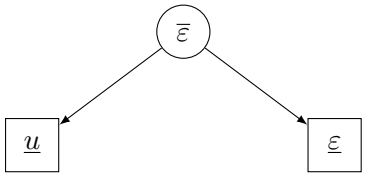
– si $\varepsilon \in L$, alors il existe $u \in \Sigma^*$ tel que $u \notin L$ (car $L \neq \Sigma^*$). Un tel mot $u$ peut être obtenu en faisant un appel à `Oracle.equiv a`, où `a` est un automate à un seul état qui est initial et final, avec les transitions qui bouclent sur lui-même (c'est-à-dire reconnaissant $\Sigma^*$).

On construit alors le crible :




– si $\varepsilon \notin L$, alors il existe $u \in \Sigma^*$ tel que $u \in L$ (car $L \neq \emptyset$). Un tel mot $u$ peut être obtenu en faisant un appel à `Oracle.equiv a`, où `a` est un automate à un seul état qui est initial **mais non final**, avec les transitions qui bouclent sur lui-même (c'est-à-dire reconnaissant $\emptyset$).

On construit alors le crible :




**Question 24** On fait ici l'hypothèse que $c_\gamma \sigma_\gamma = \varepsilon$ (sinon les termes sont mal définis). On a $\tau_\gamma = p_\gamma = \lfloor c \rfloor$ et $\hat{\tau}_\gamma = \hat{p}_\gamma = \delta^*(\underline{\varepsilon}, c)$.

Distinguons :

– si $c \in L$, alors l'automate associé à $T$ ne reconnaît pas $c$ (car $c$ est un contre-exemple), et donc $\delta^*(\underline{\varepsilon}, c)$ est une feuille dans l'enfant droit de $T$. Par ailleurs, comme $c \in L$, lorsqu'on crible $c$, on commence par le côté gauche. On en déduit que $\lfloor c \rfloor$ est une feuille dans l'enfant gauche.

– on raisonne symétriquement si $c \notin L$.

Dans les deux cas, l'un est gauche et l'autre à droite. Par ailleurs, comme $T$ est accessible, pour chaque feuille $w$, $\lfloor w \rfloor = \underline{w}$. Comme $T$ est démêlant, le premier test d'appartenance obtenu en criblant un mot $w$ est le test $w \in L$. On en déduit qu'entre $\tau_\gamma = p_\gamma$ et $\hat{\tau}_\gamma$, l'un est dans $L$ (celui à gauche) et pas l'autre.

**Question 25** Il y a peut-être une manière efficace de faire le calcul, mais comme aucune complexité n'est imposée, on se propose d'appliquer la définition pour les calculs des différents mots.

On commence par écrire une première fonction `cut` telle que `cut i c` renvoie le triplet $(\pi_i, c_i, \sigma_i)$.

```
let rec cut i c = match i, c with
    | 0, c_0 :: c' -> [], c_0, c'
    | gamma, x :: c' ->
        let pi_i, c_i, sigma_i = cut (i - 1) c' in
        x :: pi_i, c_i, sigma_i
    | _ -> failwith "Erreur"
```

Ensuite, on peut appliquer la définition pour trouver $j$, en utilisant les fonctions écrites précédemment.

```
let split t c =
    let a = automaton_of_disctree t in
    let i = ref 0 and trouve = ref false in
    while not !trouve do
        let pi_i, c_i, sigma_i = cut !i c in
        let p_i = sift t pi_i and
            hat_p_i = delta_star a [] pi_i in
        if cmp_word p_i hat_p_i <> 0 then trouve := true
        else incr i
    done;
    let pi_j, c_j, sigma_j = cut !i c in
    let p_j = sift t pi_j and p_j_plus_un = sift t (pi_j @ [c_j]) in
    p_j @ [c_j], p_j_plus_un, sigma_j
```

**Question 26** Il y a deux choses à faire ici : trouver le nœud $\underline{p_{j+1}}$ à remplacer, et déterminer l'ordre des enfants. Pour le deuxième point, il suffit de tester si $p_{j+1}\sigma_j \in L$ (auquel cas $\underline{p_{j+1}}$ est à gauche) ou non (auquel cas il est à droite).

Pour le premier point, on propose une recherche naïve dans l'arbre.

```
let substitute t (pp, p, s) =
    let new_node =
        if Oracle.mem (p @ s) then
            Node (Leaf p, s, Leaf pp)
        else
            Node (Leaf pp, s, Leaf p)
    in
    let rec replace = function
        | Leaf v ->
            if cmp_word v p = 0 then new_node
            else Leaf v
        | Node (l, v, r) ->
            Node (replace l, v, replace r)
    in
    replace t
```

**Question 27** Comme aucune spécification de fonction n'est indiquée, on suppose qu'on attend ici une description de haut niveau en français par exemple. On propose l'algorithme suivant :

- Initialiser un crible $T$ avec la question 23.
- Tant que l'automate associé à $T$ ne reconnaît pas $L$ :
  - Soit $c$ un contre-exemple renvoyé par l'oracle.
  - Calculer le triplet $(p_j c_j, p_{j+1}, \sigma_j)$ à l'aide de la question 25.
  - Remplacer $T$ par l'arbre obtenu à l'aide de la question 26.
- Renvoyer l'automate associé à $T$.

On peut rajouter des premiers tests d'équivalence avec un automate à un seul état pour vérifier si $L = \emptyset$ ou $L = \Sigma^*$.

Le nombre de feuilles de $T$ est un variant strictement croissant, et on a montré à la question 20 qu'il était borné.

**Question 28** Notons $A = (Q, \underline{\varepsilon}, \delta, \mathbb{1}_F)$ l'automate renvoyé par l'algorithme précédent.

Raisonnons par l'absurde et supposons qu'il existe un automate $B = (Q_B, q_0, \delta_B, \mathbb{1}_{F_B})$ reconnaissant $L$ tel que $|Q_B| < |Q|$.

Par principe des tiroirs et par accessibilité, il existe $u_1, u_2 \in \Sigma^*$ tels que :

- $\delta^*(\underline{\varepsilon}, u_1) \neq \delta^*(\underline{\varepsilon}, u_2)$ ;
- $\delta_B^*(q_0, u_1) = \delta_B^*(q_0, u_2)$.

Alors par la question 18, $u_1$ et $u_2$ ayant des criblats différents dans le crible $T$ final, on en déduit qu'il sont séparables. Il existe donc $v \in \Sigma^*$ tel que $\mathbb{1}_L(u_1 v) \neq \mathbb{1}_L(u_2 v)$.

Mais alors $\delta_B^*(q_0, u_1 v) = \delta_B^*(q_0, u_2 v)$, donc $\mathbb{1}_L(u_1 v) = \mathbb{1}_L(u_2 v)$, ce qui est absurde.

On en déduit que l'automate $A$ a bien un nombre minimal d'états.