Concours commun Mines-Ponts, France

Session 2025, filière MP

Informatique option MP

Durée : 3 heures

Ce que traite le sujet

- Section 1 : alphabets, mots et automates
- Section 2 : relation de séparabilité
- Section 3 : arbre discriminant
- Section 4 : automate tiré d'un arbre discriminant

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/mines/mp/2025/info-option

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

A2025 – INFO MP

ÉCOLE NATIONALE DES PONTS et CHAUSSÉES,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.

Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).

CONCOURS 2025

ÉPREUVE D'INFORMATIQUE MP

Durée de l'épreuve : 3 heures

L'usage de la calculatrice ou de tout dispositif électronique est interdit.

Les candidats sont priés de mentionner de façon apparente sur la première page de la copie :

INFORMATIQUE - MP

Cette épreuve concerne uniquement les candidats de la filière MP.

L'énoncé de cette épreuve comporte 10 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

.

Les sujets sont la propriété du GIP CCMP. Ils sont publiés sous les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France.

Tout autre usage est soumis à une autorisation préalable du Concours commun Mines Ponts.

Préliminaires

L'épreuve est formée d'un problème unique, constitué de 28 questions, et porte sur l'apprentissage automatique (ou *machine learning*) d'un langage régulier que l'on souhaite déterminer à partir de requêtes d'appartenance et de requêtes d'équivalence.

Les applications de l'apprentissage automatique d'un langage sont étendues. En analysant le modèle appris, il est possible de détecter les divergences entre une spécification et sa mise en œuvre ou entre différentes mises en œuvre tant pour des systèmes matériels que logiciels : par exemple, protocoles de cartes à puce, réseaux de zombies sur internet (ou *botnets*), logiciels patrimoniaux (ou *legacy software*) pour ne citer que quelques illustrations.

Le problème est divisé en quatre sections reliées entre elles. Une question peut être traitée à condition d'avoir lu les définitions introduites jusque là. Dans la première section (page 2), nous établissons quelques prolégomènes. Dans la deuxième section (page 4), nous nous intéressons à une relation d'équivalence induite par le langage à apprendre. Dans la troisième section (page 6), nous étudions un certain type d'arbre de décision. Dans la quatrième section (page 8), nous construisons un automate reconnaissant le langage à apprendre.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères différentes désigne la même entité mais du point de vue mathématique pour la police en italique (par exemple n, T, δ) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`, `t`, `delta`).

Des rappels des extraits du manuel de documentation de OCaml portant sur le module `List` sont reproduits en annexe (page 10).

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Il faudra coder des fonctions à l'aide du langage de programmation OCaml exclusivement, en reprenant l'en-tête de fonctions fourni par le sujet, sans s'obliger à recopier la déclaration des types. Il est permis d'utiliser la totalité du langage OCaml sauf indication contraire. Il est recommandé de s'en tenir aux fonctions les plus courantes afin de rester compréhensible. Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté et de la concision des programmes : il est attendu que l'on choisisse des noms de variables intelligibles ou encore que l'on structure de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1. Alphabets, mots et automates

Dans l'ensemble du sujet, nous fixons un *alphabet* binaire $\Sigma = \{a, b\}$ muni de l'ordre alphabétique ; la notation ε désigne le *mot vide* ; la notation Σ^* désigne l'*ensemble des mots* sur l'alphabet Σ muni de l'ordre lexicographique induit. La *concaténation* de deux mots $u \in \Sigma^*$ et $v \in \Sigma^*$ est notée uv.

Indication OCaml : Nous adoptons les types suivants :

```
1. type letter = A | B
2. type word = letter list
```

☐ 1 – Écrire une fonction OCaml `cmp_letter (x1:letter) (x2:letter) : int` dont la valeur de retour est nulle si les lettres x_1 et x_2 sont égales, est strictement négative si la lettre x_1 précède strictement la lettre x_2 dans l'ordre alphabétique et est strictement positive sinon.

☐ 2 – Écrire une fonction OCaml `cmp_word (w1:word) (w2:word) : int` dont la valeur de retour est nulle si les mots w_1 et w_2 sont égaux, est strictement négative si le mot w_1 précède strictement le mot w_2 dans l'ordre lexicographique et est strictement positive sinon. On s'interdira l'usage de la fonction `compare` du module `List` et toute autre approche similaire.

☐ 3 – Décrire une structure de données immuables qui implémente un dictionnaire dont les clés sont des mots de Σ^* et dont les valeurs sont d'un type quelconque. Il est attendu un nom de structure classique, un type OCaml, une condition concernant l'ensemble des clés et un invariant de bonne constitution.

Indication OCaml : Dans la suite du sujet, nous nous munissons d'un module OCaml `WordMap` et d'un type `'a wordmap` qui implémentent des dictionnaires de clés de type `word` et de valeurs d'un type quelconque `'a`. Nous disposons de la constante et des deux fonctions suivantes :

- `WordMap.empty`, de type `'a wordmap`, qui désigne le dictionnaire vide.
- `WordMap.add`, de type `word -> 'a -> 'a wordmap -> 'a wordmap`, telle que la valeur de retour de `WordMap.add w v d` est un dictionnaire contenant les associations du dictionnaire d ainsi qu'une association supplémentaire entre la clé w et la valeur v.
- `WordMap.find`, de type `word -> 'a wordmap -> 'a`, telle que la valeur de retour de `WordMap.find w d` est la valeur associée à la clé w dans le dictionnaire d.

☐ 4 – À titre d'exemple, déclarer deux valeurs OCaml `delta_a` et `delta_b`, de type `word wordmap`, et une valeur OCaml `un_a`, de type `bool wordmap`, égales aux dictionnaires respectifs

$$\delta_a : \begin{cases} \Sigma^* & \to & \Sigma^* \\ \varepsilon & \mapsto & a \\ a & \mapsto & \varepsilon \end{cases} \quad ; \quad \delta_b : \begin{cases} \Sigma^* & \to & \Sigma^* \\ \varepsilon & \mapsto & \varepsilon \\ a & \mapsto & a \end{cases} \quad \text{et} \quad \mathbb{1}_{\{a\}} : \begin{cases} \Sigma^* & \to & \{\mathsf{faux}, \mathsf{vrai}\} \\ \varepsilon & \mapsto & \mathsf{faux} \\ a & \mapsto & \mathsf{vrai} \end{cases}.$$

Définition : Le terme *automate* s'entend systématiquement dans le sens d'un automate fini déterministe complet, c'est-à-dire un quadruplet $(Q, q_0, \delta, \mathbb{1}_F)$ où

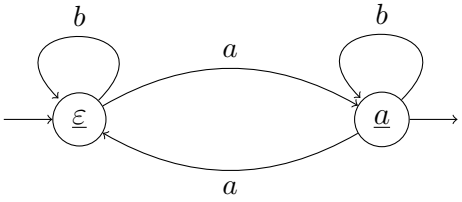
— l'ensemble Q est un ensemble fini d'états,

— l'état q_0 est un élément particulier de Q appelé l'état initial,

— l'application δ est une application $Q \times \Sigma \to Q$ appelée fonction de transition,

— et l'application $\mathbb{1}_F : Q \to \{\mathsf{faux}, \mathsf{vrai}\}$ est la fonction indicatrice d'une partie F de Q appelée ensemble des états finals.

☐ 5 – À titre d'exemple, décrire en langue française le langage reconnu par l'automate figuré ci-dessous puis énoncer une expression régulière qui le dénote. On ne demande pas de justification.



☐ 6 – Dire s'il existe une expression régulière admettant l'automate figuré à la question 5 comme automate de Glushkov associé.

Indication OCaml : Dans ce sujet, l'ensemble des états des automates est systématiquement choisi parmi les mots de Σ^*. Typographiquement, nous convenons de souligner les mots servant d'état dans ce sujet et adoptons les types OCaml suivants

```
3.  type state = word
4.  type automaton = { initial : state ;
5.                     transitions : state -> letter -> state ;
6.                     finals : state -> bool }
```

☐ 7 – À titre d'exemple, définir une valeur OCaml `automaton_example`, de type `automaton`, égale à l'automate figuré à la question 5.
On copiera et on complètera le code suivant :

```
1.  let automaton_example =
2.  {initial =  ....  ;
3.   transitions =  ( fun q x -> WordMap.find q  (.....)
4.                  );
5.   finals = .....
6.  }
```

Définition : Pour tout automate $A = (Q, q_0, \delta, \mathbb{1}_F)$, pour tout état q de Q et pour tout mot w de Σ^*, nous notons $\delta^*(q, w)$ l'état d'arrivée dans l'automate A au terme du parcours démarrant en l'état q et suivant les transitions de A étiquetées par les lettres successives du mot w. Ainsi, la fonction δ^* est une application $Q \times \Sigma^* \to Q$ qui étend l'application δ.

☐ 8 – Écrire une fonction OCaml `delta_star (a:automaton) (q:state) (w:word) : state` dont la valeur de retour est $\delta^*(q, w)$.

☐ 9 – Écrire une fonction OCaml `accepts (a:automaton) (w:word) : bool` dont la valeur de retour est le booléen vrai si le mot w est reconnu par l'automate A et le booléen faux sinon. Il est demandé d'utiliser la fonction de la question précédente.

2. Relation de séparabilité par rapport à un langage

Il a été fixé, jusqu'à la fin de ce sujet, un certain langage régulier $L \subseteq \Sigma^*$, que nous ne connaissons pas mais que nous souhaitons apprendre. Autrement dit, notre objectif est de produire en temps fini et efficacement un automate qui reconnait le langage L. Nous accédons au langage L par l'intermédiaire d'un oracle qui répond à deux types de requêtes :

— des *requêtes d'appartenance*, prenant la forme de la question « le mot $w \in \Sigma^*$ appartient-il au langage fixé L ? »

— et des *requêtes d'équivalence*, prenant la forme de la question : « l'automate A reconnait-il le langage fixé L ? ».

En cas de réponse négative à une requête d'équivalence, l'oracle nous pourvoit d'un contre-exemple $c \in \Sigma^*$ pour lequel l'automate A se trompe. Autrement dit, soit le mot c appartient au langage à apprendre L mais n'est pas reconnu par l'automate A, soit le mot c n'appartient pas au langage à apprendre L mais est reconnu par l'automate A.

Indication OCaml : L'oracle est empaqueté dans un module `Oracle` ainsi signé :

```
7.  type answer = Yes | Counterexample of word
8.
9.  val mem : word -> bool
10. val equiv : automaton -> answer
```

et s'utilise avec la syntaxe `Oracle.mem w` pour une requête d'appartenance ou `Oracle.equiv a` pour une requête d'équivalence.

Définition : Nous notons $\mathbb{1}_\Lambda : \Sigma^* \to \{\text{faux}, \text{vrai}\}$ la *fonction indicatrice* de tout langage $\Lambda \subseteq \Sigma^*$. Pour tout mot $u \in \Sigma^*$, $\mathbb{1}_\Lambda(u)$ prend la valeur vrai si et seulement si le mot u appartient au langage Λ.

Définition : Deux mots quelconques $u_1 \in \Sigma^*$ et $u_2 \in \Sigma^*$ sont dits *séparés par le mot* $\overline{v}$ *par rapport à un langage* $\Lambda \subseteq \Sigma^*$ si l'on a

$$\mathbb{1}_\Lambda(u_1\overline{v}) \neq \mathbb{1}_\Lambda(u_2\overline{v}),$$

autrement dit, si concaténation $u_1 v$ appartient au langage Λ tandis que la concaténation $u_2\overline{v}$ n'appartient pas au langage Λ ou si la concaténation $u_1\overline{v}$ n'appartient pas au langage Λ tandis que la concaténation $u_2\overline{v}$ appartient au langage Λ. Nous disons alors que le mot $\overline{v}$ est un *discriminant*.

Indication OCaml : Typographiquement, nous convenons de surligner les mots servant de discriminant dans ce sujet et marquons le rôle de discriminant en adoptant l'alias de type

```
11.  type disc = word
```

☐ 10 – Écrire une fonction OCaml `separated_by (u1:word) (u2:word) (v:disc) : bool` qui teste si les deux mots u_1 et u_2 sont séparés par rapport au langage à apprendre L par le mot discriminant $\overline{v}$.

Définitions : Deux mots quelconques $u_1 \in \Sigma^*$ et $u_2 \in \Sigma^*$ sont dits *séparables par rapport à un langage* $\Lambda \subseteq \Sigma^*$, ou simplement *séparables*, s'il existe un discriminant qui les sépare. Ils sont dits *inséparables* sinon. Nous notons $u_1 \equiv_\Lambda u_2$ la relation d'inséparabilité par rapport au langage Λ.

☐ 11 – Montrer que la relation d'inséparabilité par rapport à un langage quelconque est une relation d'équivalence sur l'ensemble des mots Σ^*.

☐ 12 – Soient $A = (Q, q_0, \delta, \mathbb{1}_F)$ un automate quelconque, $L_A \subseteq \Sigma^*$ le langage reconnu par l'automate A et $(u_1, u_2) \in (\Sigma^*)^2$ deux mots tels que $\delta^*(q_0, u_1) = \delta^*(q_0, u_2)$. Montrer que les mots u_1 et u_2 sont inséparables par rapport au langage L_A.

☐ 13 – Citer un théorème liant langages réguliers d'une part et langages reconnus par un automate d'autre part. Déduire de la question 12 que le nombre de classes d'équivalences de la relation d'inséparabilité par rapport au langage à apprendre L est fini.

3. Arbre discriminant

Définition : Un *arbre discriminant* est un arbre de décision binaire, dont les sommets internes sont étiquetés par des discriminants et dont les feuilles sont étiquetées par des états distincts deux à deux.

```
12.  type disctree =   Node of disctree * disc * disctree
13.                  | Leaf of state
```

La figure 1 présente un exemple d'arbre discriminant $\mathcal{T}_0$, dans lequel les trois sommets internes sont figurés par des cercles et les quatre feuilles par des rectangles.

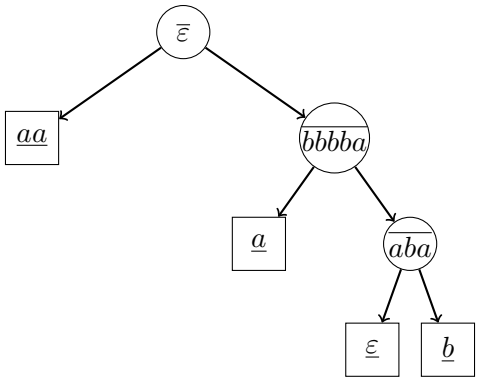


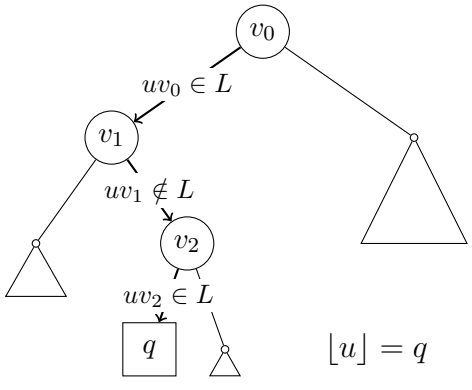
FIGURE 1 – Un exemple d'arbre discriminant : l'arbre $\mathcal{T}_0$

☐ 14 – Écrire une fonction OCaml `states_of_disctree (t:disctree) : state list` dont la valeur de retour est une liste des étiquettes des feuilles de l'arbre discriminant T. Il est attendu que la complexité en temps de `states_of_disctree` soit linéaire en le nombre de feuilles et qu'on le justifie.
Par exemple, avec l'arbre discriminant $\mathcal{T}_0$ de la figure 1, la valeur `states_of_disctree t0` peut égaler la liste $[aa, a, \varepsilon, b]$, la liste $[\varepsilon, a, aa, b]$ ou encore toute autre permutation.

Définition : *Cribler un mot $u \in \Sigma^*$ à travers un arbre discriminant T pour un langage L* consiste à construire l'unique chemin $s_0 - s_1 - \cdots - s_k$ dans l'arbre T où :

- le sommet s_0 est la racine de T,
- pour tout indice i compris entre 0 et $k-1$, le sommet s_i est un sommet interne et, en notant v_i l'étiquette du sommet interne s_i, si le mot concaténé uv_i appartient au langage à apprendre L, alors le sommet s_{i+1} est le fils gauche de s_i, sinon, le sommet s_{i+1} est le fils droit de s_i,
- le sommet s_k est une feuille de l'arbre T.

et renvoyer l'état q porté par la feuille s_k. L'état q s'appelle le *criblat du mot u* et est noté $\lfloor u \rfloor$.



Par exemple, en supposant que le langage à apprendre L est dénoté par l'expression régulière ab^*a, le criblat du mot $abbba$ par l'arbre discriminant $\mathcal{T}_0$ (figure 1) est l'état $\lfloor abbba \rfloor = \underline{aa}$, qui est l'extrémité du chemin $\overline{\varepsilon} - \underline{aa}$ obtenu en observant que $abbba$ appartient à L. En revanche, le criblat du mot abb est l'état $\lfloor abb \rfloor = \underline{a}$, qui est l'extrémité du chemin $\overline{\varepsilon} - \overline{bbbba} - \underline{a}$ obtenu en observant que abb appartient à L mais $abbbbbba$ n'appartient pas à L.

☐ 15 – Donner le criblat du mot ba pour l'exemple précédent.

☐ 16 – Écrire une fonction OCaml `sift (t:disctree) (u:word) : state` dont la valeur de retour est le criblat $\lfloor u \rfloor$ du mot u à travers l'arbre discriminant T pour le langage L.

☐ 17 – Déterminer la complexité en temps dans le pire des cas de la fonction `sift t u` en fonction de tout ou partie des grandeurs suivantes : la longueur $|u|$ du mot u, le nombre n de sommets de l'arbre T, la hauteur h de l'arbre T, le maximum $M = \max_{v \in T} |v|$ des longueurs des mots discriminants de l'arbre T, le coût $\mu(|w|)$ d'un appel `Oracle.mem w`, le coût $\nu(|\mathcal{A}|)$ d'un appel `Oracle.equiv a`.

☐ 18 – Démontrer, en exhibant un discriminant, que deux mots de criblats distincts sont toujours séparables.

☐ 19 – Démontrer que deux mots inséparables ont les mêmes criblats.

Définition : Nous disons qu'un arbre discriminant est *accessible* si, pour tout état $\underline{w} \in \Sigma^*$, apparaissant dans la liste des étiquettes des feuilles, le criblat $\lfloor w \rfloor$ du mot w est l'état $\underline{w}$.

☐ 20 – Démontrer qu'il existe une constante dépendant uniquement du langage à apprendre L, qui majore le nombre de feuilles d'un arbre discriminant accessible.

Définition : Nous disons qu'un arbre discriminant est *démêlant* s'il comporte au moins un sommet interne et si l'étiquette de la racine est le discriminant mot vide $\overline{\varepsilon}$.

☐ 21 – Démontrer que si deux mots u_1 et $u_2 \in \Sigma^*$ ont le même criblat par rapport à un arbre discriminant démêlant, alors on a l'égalité $\mathbb{1}_L(u_1) = \mathbb{1}_L(u_2)$.

4. Automate tiré d'un arbre discriminant

Définitions : Nous disons qu'un arbre discriminant T est un *crible* s'il est accessible, démêlant et si le mot vide $\underline{\varepsilon}$ est un état qui apparait dans la liste des étiquettes des feuilles.

L'*automate associé à un crible T pour un langage L* est l'automate A ainsi défini :

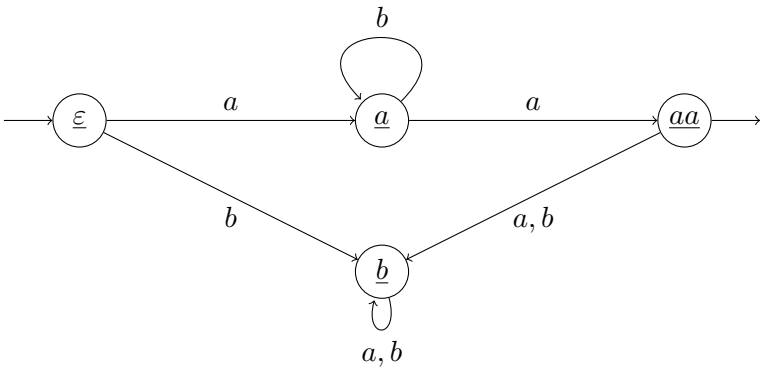
— l'ensemble des états de A est l'ensemble des étiquettes des feuilles de T ;

— l'état initial de A est l'état $\underline{\varepsilon}$;

— la fonction de transition associe l'état $\underline{w}$ et la lettre x à l'état $\lfloor wx \rfloor$ obtenu en criblant le mot wx à travers l'arbre discriminant T pour le langage L ;

— l'ensemble des états finals est l'ensemble des mots $\underline{w}$ où $\underline{w}$ est une étiquette d'une feuille du sous-arbre gauche de T.

Par exemple, l'automate associé au crible $\mathcal{T}_0$ de la figure 1 pour le langage dénoté par ab^*a est l'automate ainsi figuré



☐ 22 – Écrire une fonction OCaml `automaton_of_disctree (t:disctree) : automaton` dont la valeur de retour est l'automate associé au crible T.
On recopiera et on complètera le code suivant :

```
7.  let automaton_of_disctree (t:disctree) : automaton =
8.    { initial =  .... ;
9.      transitions =  .... ;
10.     finals =  ....
11.   }
```

☐ 23 – En supposant que le langage à apprendre L n'est ni le langage vide, ni le langage plein Σ^*, décrire une construction, à l'aide des fonctions `Oracle.mem` et `Oracle.equiv`, d'un premier crible formé d'un seul sommet interne et de deux feuilles.

Dans les questions 24, 25 et 26, nous notons $A = (Q, \underline{\varepsilon}, \delta, \mathbb{1}_F)$ l'automate associé à un certain crible T déjà construit. Nous supposons que `Oracle.equiv a` renvoie `Oracle.Counterexample c`, où c est le mot non vide $c = c_0c_1 \cdots c_{\gamma-1} \in \Sigma^*$. Nous nous proposons de construire un nouveau crible T' ayant un état de plus que le crible T.

Pour tout indice i compris entre 0 et γ, nous appelons $\pi_i = c_0 c_1 \cdots c_{i-1}$ le préfixe de c de longueur i et $\sigma_i = c_{i+1} c_1 \cdots c_{\gamma-1}$ le suffixe de longueur $\gamma - i - 1$ de sorte que le mot c admette la factorisation $c = \pi_i c_i \sigma_i$. Nous notons les états

$$\underline{p_i} = \lfloor \pi_i \rfloor \in \Sigma^* \qquad \text{et} \qquad \underline{\hat{p}_i} = \delta^*(q_\varepsilon, \pi_i) \in \Sigma^*$$

et posons encore

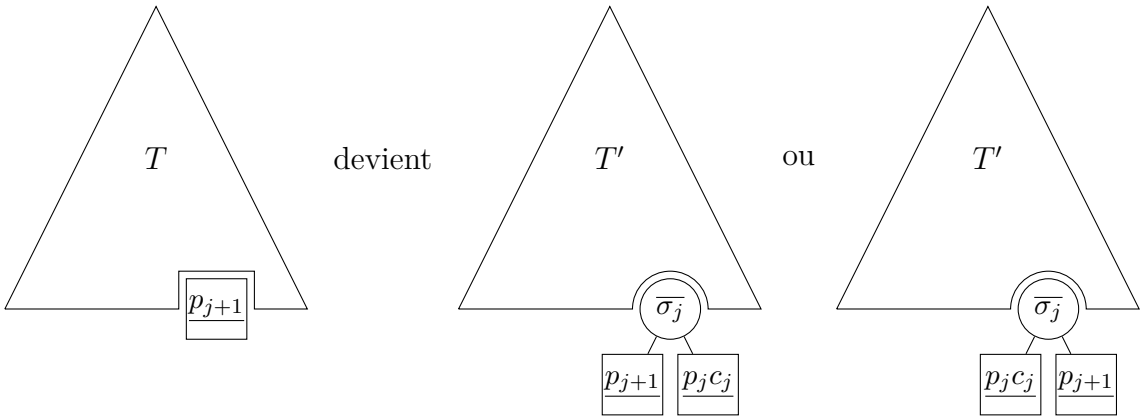
$$\tau_i = p_i c_i \sigma_i \in \Sigma^* \qquad \text{et} \qquad \hat{\tau}_i = \hat{p}_i c_i \sigma_i \in \Sigma^*.$$

☐ 24 – Démontrer que l'on a $\mathbb{1}_L(\tau_\gamma) \neq \mathbb{1}_L(\hat{\tau}_\gamma)$.

On admet qu'il existe un indice i compris entre 0 et $\gamma - 1$ tel que les états $\underline{p_i}$ et $\underline{\hat{p}_i}$ sont égaux, les états $\underline{p_{i+1}}$ et $\underline{\hat{p}_{i+1}}$ sont distincts et, alors, les mots $p_i c_i$ et p_{i+1} sont séparés par le discriminant $\overline{\sigma_i}$.

☐ 25 – Écrire une fonction OCaml `split (t:disctree) (c:word) : state * state * disc` dont la valeur de retour est le triplet $(\underline{p_j c_j}, \underline{p_{j+1}}, \overline{\sigma_j})$ où l'entier j est le plus petit des indices i compris entre 0 et $\gamma - 1$ obtenu par la question 24.

On admet de la question 24 que l'arbre discriminant T' obtenu en substituant un arbre constitué d'un sommet interne d'étiquette $\overline{\sigma_j}$, d'une feuille d'étiquette l'ancien état $\underline{p_{j+1}}$ et d'une feuille d'étiquette un nouvel état $\underline{p_j c_j}$ à la place de la feuille d'étiquette $\underline{p_{j+1}}$ dans l'arbre discriminant T jouit encore des propriétés de crible. Schématiquement,



☐ 26 – Écrire une fonction OCaml `substitute (t:disctree) ((pp,p,s):state * state * disc) : disctree` ainsi spécifiée :
Précondition : Le triplet (pp, p, s) est la valeur de retour de la fonction `split` écrite à la question 25.
Valeur de retour : Crible T' décrit ci-dessus.

☐ 27 – Concevoir un algorithme complet qui apprend un langage régulier à base de requêtes d'appartenance et de requêtes d'équivalence en renvoyant un automate. Justifier sa terminaison à l'aide d'un variant de boucle.

☐ 28 – Démontrer que l'automate construit à la question 27 possède un nombre d'état minimal parmi l'ensemble des automates qui reconnaissent le langage à apprendre.

A. Annexe : aide à la programmation en OCaml

Nous reproduisons des extraits de la documentation OCaml.

Opérations sur les listes : Le module `List` offre les fonctions suivantes :

- `length : 'a list -> int`
  Return the length (number of elements) of the given list.
- `append : 'a list -> 'a list -> 'a list`
  `append l0 l1` appends `l1` to `l0`. Same function as the infix operator `@`.
- `iter : ('a -> unit) -> 'a list -> unit`
  `iter f [a1; ...; an]` applies function `f` in turn to `[a1; ...; an]`. It is equivalent to `f a1; f a2; ...; f an`.
- `map : ('a -> 'b) -> 'a list -> 'b list`
  `map f [a1; ...; an]` applies function `f` to `a1`, ..., `an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.
- `fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc`
  `fold_left f init [b1; ...; bn]` is `f (... (f (f init b1) b2) ...) bn`.
- `for_all : ('a -> bool) -> 'a list -> bool`
  `for_all f [a1; ...; an]` checks if all elements of the list satisfy the predicate `f`. That is, it returns `(f a1) && (f a2) && ... && (f an)` for a non-empty list and `true` if the list is empty.
- `exists : ('a -> bool) -> 'a list -> bool`
  `exists f [a1; ...; an]` checks if at least one element of the list satisfies the predicate `f`. That is, it returns `(f a1) || (f a2) || ... || (f an)` for a non-empty list and `false` if the list is empty.
- `mem : 'a -> 'a list -> bool`
  `mem a set` is true if and only if `a` is equal to an element of `set`.
- `filter : ('a -> bool) -> 'a list -> 'a list`
  `filter f l` returns all the elements of the list `l` that satisfy the predicate `f`. The order of the elements in the input list is preserved.

D'après `https://ocaml.org/manual/5.1/api/List.html`

Fin de l'épreuve

Travailler les chapitres de ce sujet

Prepa Help suit le programme de MP chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Langage OCaml : expressions, types, filtrage, récursivité
- Dictionnaires et principes du hachage
- Logique, ensembles et applications
- Complexité des algorithmes itératifs et récursifs
- Spécifier, tester, prouver : terminaison et correction

Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière : **écrire sur WhatsApp**

Les autres années et épreuves : prepahelp.com/annales/mines/mp