A 2015 INFO. MP

École des Ponts ParisTech,
SUPAERO (ISAE), ENSTA ParisTech,
Télécom ParisTech, Mines ParisTech,
Mines de Saint-Étienne, Mines Nancy,
Télécom Bretagne, ENSAE ParisTech (filière MP),
École polytechnique (filière TSI)

Concours 2015

Épreuve d'Informatique

Filière : MP

Durée de l'épreuve : 3 heures.

L'utilisation d'une calculatrice est autorisée.

Sujet mis à la disposition des concours :
Cycle international, Écoles des Mines, Télécom Sud Paris, TPE-EIVP.

L'énoncé de cette épreuve comporte 10 pages.

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

Recommandations aux candidats

- **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.**
- **Tout résultat fourni dans l'énoncé peut être utilisé pour les questions ultérieures même s'il n'a pas été démontré.**
- **Il ne faut pas hésiter à formuler les commentaires qui semblent pertinents même lorsque l'énoncé ne le demande pas explicitement.**

Composition de l'épreuve

L'épreuve comporte un unique problème (pages 2 à 10), comportant 32 questions.

Problème : Automates d'arbre

Préliminaire concernant la programmation

Il faudra coder des fonctions à l'aide du langage de programmation Caml, tout autre langage étant exclu. Lorsque le candidat écrira une fonction, il pourra faire appel à d'autres fonctions définies dans les questions précédentes ; il pourra aussi définir des fonctions auxiliaires. Quand l'énoncé demande de coder une fonction, il n'est pas nécessaire de justifier que celle-ci est correcte, sauf si l'énoncé le demande explicitement. Enfin, si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien vérifiées.

Dans les énoncés du problème, un même identificateur écrit dans deux polices de caractères différentes désignera la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

Fonctions utilitaires

Dans cette partie, on code quelques fonctions générales qui seront utiles par la suite. On ne cherchera pas à proposer l'implémentation la plus efficace possible de chaque fonction.

Quand il est question de donner la complexité d'une fonction, il s'agit de calculer la complexité asymptotique en temps, en notation $O(\cdot)$, de cette fonction dans le pire des cas. Il est inutile de donner une preuve de cette complexité.

☐ 1 – Coder une fonction Caml `contient: 'a list -> 'a -> bool` telle que `contient li x` renvoie un booléen qui vaut *Vrai* si et seulement si la liste `li` contient l'élément `x`. Donner la complexité de cette fonction.

☐ 2 – En utilisant la fonction `contient`, coder une fonction Caml `union: 'a list -> 'a list -> 'a list` telle que `union l1 l2`, où `l1` et `l2` sont deux listes d'éléments sans doublon dans un ordre arbitraire, renvoie une liste sans doublon contenant l'union des éléments des deux listes, dans un ordre arbitraire. Donner la complexité de cette fonction.

☐ 3 – En utilisant la fonction `union`, coder une fonction Caml `fusion: 'a list list -> 'a list` telle que `fusion l`, où `l` est une liste de listes d'éléments, chacune de ces listes étant sans doublon, renvoie une liste de tous les éléments contenus dans au moins une des listes de la liste `l`, sans doublon et dans un ordre arbitraire. En notant $l = (l_1, \ldots, l_k)$ la liste codée par `l` et en posant $L \coloneqq \sum_{j=1}^{k} |l_j|$, donner la complexité de la fonction `fusion` en fonction de L.

☐ 4 – Coder `produit: 'a list -> 'b list -> ('a * 'b) list`, telle que `produit l1 l2` renvoie une liste de tous les couples `(x,y)` avec `x` un élément de `l1` et `y` un élément de `l2`. On supposera les listes `l1` et `l2` sans doublon. La liste résultante doit avoir pour longueur le produit des longueurs des deux listes. Donner la complexité de cette fonction.

Arbres binaires étiquetés

Soit $\Sigma = \{\alpha_0, \ldots, \alpha_{m-1}\}$ un ensemble fini non vide de m symboles, appelé *alphabet*. En Caml, on représentera le symbole α_k (pour $0 \leqslant k \leqslant m-1$) par l'entier k. Cet alphabet sera supposé fixé dans tout l'énoncé.

Un *arbre binaire* étiqueté par Σ (simplement appelé, dans ce problème, *arbre*) est soit *l'arbre vide* (noté ε), soit un quintuplet (S, r, λ, g, d), où :

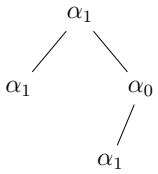
(i) S est un ensemble fini non vide dont les éléments sont appelés *nœuds* ;
(ii) $r \in S$ est la *racine* de S ;
(iii) $\lambda : S \to \Sigma$ est une application associant à chaque nœud de S une *étiquette* de Σ ;
(iv) $g : S_g \to S\backslash\{r\}$, où $S_g \subseteq S$, est une application injective associant à un nœud u de S_g un nœud appelé *fils gauche* de u ;
(v) $d : S_d \to S\backslash\{r\}$, où $S_d \subseteq S$, est une application injective associant à un nœud u de S_d un nœud appelé *fils droit* de u.

On dit qu'un nœud v est *descendant* d'un nœud u s'il existe une séquence de nœuds $u_0, u_1, \ldots, u_p \in S$ avec $p > 0$ telle que $u_0 = u$, $u_p = v$ et, pour tout $0 \leqslant k \leqslant p-1$, soit $u_{k+1} = g(u_k)$, soit $u_{k+1} = d(u_k)$.

On requiert que tout nœud sauf la racine soit le fils gauche ou droit d'un unique nœud, et qu'aucun nœud ne soit à la fois fils gauche et fils droit :

$$\forall u \in S\backslash\{r\}, |g^{-1}(\{u\})| + |d^{-1}(\{u\})| = 1.$$

Par ailleurs, tout nœud de $S\backslash\{r\}$ doit être descendant de r.



Un arbre admet une représentation graphique naturelle. Par exemple, l'arbre $t_0 = (\{u_0, u_1, u_2, u_3\}, u_0, \lambda, g, d)$ est représenté ci-contre, avec :
— $g(u_0) = u_1$, $g(u_2) = u_3$ et $d(u_0) = u_2$;
— $\lambda(u_0) = \lambda(u_1) = \lambda(u_3) = \alpha_1$ et $\lambda(u_2) = \alpha_0$.

On utilisera en Caml le type de données suivant pour coder un arbre :

```
type Arbre = Noeud of Noeud | Vide
 and Noeud = { etiquette: int; gauche: Arbre; droit: Arbre; };;
```

Dans ce codage, un arbre non vide est représenté par une instance du type `Noeud` décrivant sa racine ; un nœud est décrit par son étiquette (codée comme un entier), son fils gauche, son fils droit ; le fils gauche et le fils droit peuvent, à nouveau, être décrits par une instance du type `Noeud`, ou par le constructeur `Vide`, qui décrit leur absence.

Par exemple, l'arbre t_0 pourra être décrit par la variable `t0` comme suit :

```
let t0 = Noeud {
  etiquette=1;
  gauche=Noeud
    {etiquette=1; gauche=Vide; droit=Vide};
  droit=Noeud
    {etiquette=0;
     gauche=Noeud {etiquette=1; gauche=Vide; droit=Vide};
     droit=Vide}
};;
```

☐ 5 – Pour simplifier l'écriture d'arbres, coder en Caml une fonction `arbre` telle que si `x` représente une étiquette x et `ag` et `ad` représentent deux arbres a_g et a_d, alors `arbre x ag ad` représente un arbre dont la racine est étiquetée par x, avec pour fils gauche la racine de a_g (avec ses propres fils) et pour fils droit la racine de a_d (avec ses propres fils). Ainsi, cette fonction doit permettre de construire t_0 avec :

```
let t0 = arbre 1 (arbre 1 Vide Vide)
                 (arbre 0 (arbre 1 Vide Vide) Vide);;
```

☐ 6 – Coder en Caml une fonction `taille_arbre` prenant en argument une variable `t` représentant un arbre t et renvoyant le nombre de nœuds de l'arbre t.

Langages d'arbres

Soit $\mathcal{T}^\Sigma$ l'ensemble de tous les arbres étiquetés par Σ. Un *langage d'arbres* sur un alphabet Σ est un ensemble (fini ou infini) d'arbres étiquetés par Σ, c'est-à-dire un sous-ensemble de $\mathcal{T}^\Sigma$.

On considère dans ce problème certains langages particuliers, tous définis sur l'alphabet $\{\alpha_0, \alpha_1\}$:

— L_0 est l'ensemble des arbres dont au moins un nœud est étiqueté par α_0.

— Un arbre est *complet* s'il ne contient aucun nœud ayant un seul fils (c'est-à-dire, tout nœud a un fils gauche si et seulement s'il a un fils droit) ; conventionnellement, ε est considéré comme complet. Le langage L_{complet} est l'ensemble de tous les arbres complets.

— Un arbre (S, r, λ, g, d) est un *arbre-chaîne* s'il a uniquement des fils gauches : $d^{-1}(S \backslash \{r\}) = \varnothing$; conventionnellement, ε est également un arbre-chaîne. Le langage $L_{\text{chaîne}}$ est l'ensemble de tous les arbres-chaînes.

— Un arbre (S, r, λ, g, d) est *impartial* s'il a autant de fils gauches que de fils droits, c'est-à-dire si on a $|g(S)| = |d(S)|$; conventionnellement, ε est également impartial. On note $L_{\text{impartial}}$ l'ensemble de tous les arbres impartiaux.

☐ 7 – Pour chacun des quatre langages L_0, L_{complet}, $L_{\text{chaîne}}$, $L_{\text{impartial}}$, donner (sans justification) un exemple d'arbre avec au moins deux nœuds qui appartient au langage, et un exemple d'arbre avec au moins deux nœuds qui n'y appartient pas.

☐ 8 – Démontrer que tout arbre complet est impartial, mais que la réciproque est fausse.

☐ 9 – Démontrer que tout arbre impartial non vide a un nombre impair de nœuds.

Automates d'arbres descendants déterministes

Un *automate d'arbres descendant déterministe* (ou simplement *automate descendant déterministe*) sur l'alphabet Σ est un quadruplet $\mathcal{A}^{\downarrow} = (Q, q_0, F, \delta)$ où :

(i) Q est un ensemble fini non vide dont les éléments sont appelés *états* ;
(ii) $q_0 \in Q$ est appelé *état initial* ;
(iii) $F \subseteq Q$ est un ensemble dont les éléments sont appelés *états finals* ;
(iv) $\delta : Q \times \Sigma \to Q \times Q$ est une application appelée *fonction de transition* ; pour tout $q \in Q$, pour tout $\alpha \in \Sigma$, $\delta(q, \alpha)$ est un couple d'états (q_g, q_d).

Un automate descendant déterministe $\mathcal{A}^{\downarrow} = (Q, q_0, F, \delta)$ *reconnaît* un arbre t si :

— soit $t = \varepsilon$ et $q_0 \in F$;
— soit $t = (S, r, \lambda, g, d)$ et il existe une application $\varphi : S \to Q$ avec :
  (i) $\varphi(r) = q_0$;
  (ii) pour tout $u \in S$, si $(q_g, q_d) = \delta(\varphi(u), \lambda(u))$:
    — si $g(u)$ est défini ($u \in S_g$), alors on a $\varphi(g(u)) = q_g$, sinon on a $q_g \in F$;
    — si $d(u)$ est défini ($u \in S_d$), alors on a $\varphi(d(u)) = q_d$, sinon on a $q_d \in F$.

Noter que quand une telle application φ existe, elle est nécessairement unique.

Le *langage reconnu* par un automate descendant déterministe $\mathcal{A}^{\downarrow}$, noté $\mathcal{L}(\mathcal{A}^{\downarrow})$, est l'ensemble de tous les arbres reconnus par $\mathcal{A}^{\downarrow}$.

☐ 10 – Donner un automate descendant déterministe reconnaissant le langage $L_{\text{chaîne}}$; aucune justification n'est demandée.

☐ 11 – Montrer qu'il n'existe pas d'automate descendant déterministe qui reconnaît L_0.

En Caml, un état q_i de $Q = \{q_0, \ldots, q_{n-1}\}$ est codé par l'entier i. L'ensemble des états finals F est codé par un vecteur de booléens `finals_desc` de taille n, tel que `finals_desc.(i)` contient *Vrai* si et seulement si $q_i \in F$. Enfin, les transitions sont codées par une matrice de couples d'entiers, telle que `transitions_desc.(i).(k)` est le couple `(g,d)` vérifiant $(q_g, q_d) = \delta(q_i, \alpha_k)$.

On représente ainsi en Caml un automate descendant déterministe (Q, q_0, F, δ) avec le type suivant :

```
type Automate_Descendant_Deterministe = {
  finals_desc: bool vect;
  transitions_desc: (int*int) vect vect
};;
```

☐ 12 – Pour un automate descendant déterministe $\mathcal{A}^{\downarrow} = (Q, q_0, F, \delta)$ et $q \in Q$, on note $\mathcal{A}^{\downarrow}_q$ l'automate descendant déterministe (Q, q, F, δ) identique à $\mathcal{A}^{\downarrow}$ sauf pour l'état initial. Coder une fonction `applique_desc` telle que `applique_desc add q t`, où `add` représente un automate descendant déterministe $\mathcal{A}^{\downarrow} = (Q, q_0, F, \delta)$, `q` un état $q \in Q$ et `t` un arbre $t = (S, r, \lambda, g, d)$, renvoie un booléen qui vaut *Vrai* si et seulement $\mathcal{A}^{\downarrow}_q$ reconnaît t.

☐ 13 – En utilisant `applique_desc`, coder une fonction `evalue_desc` telle que `evalue_desc add t`, où `add` représente un automate descendant déterministe $\mathcal{A}^{\downarrow}$ et `t` un arbre t, renvoie un booléen qui vaut *Vrai* si et seulement si $\mathcal{A}^{\downarrow}$ reconnaît t.

Automates descendants et langages rationnels de mots

À tout mot non vide $x = x_1 \dots x_l$ avec $x_1, \dots, x_l \in \Sigma$, on associe un arbre-chaîne $\mathit{chaîne}(x) = (\{u_1 \dots u_l\}, u_1, \lambda, g, d)$ vérifiant : pour $1 \leqslant i \leqslant l, \lambda(u_i) = x_i$ et pour $1 \leqslant i \leqslant l-1, g(u_i) = u_{i+1}$, d n'étant défini pour aucun u_i, et $g(u_l)$ étant non défini. Par convention, $\mathit{chaîne}(\varepsilon) = \varepsilon$ (où le premier ε est le mot vide, le second l'arbre vide).

Pour un langage de mots L, on définit le langage d'arbres $\mathit{chaîne}(L) \coloneqq \{\mathit{chaîne}(x) \mid x \in L\}$.

☐ 14 – Soit L un langage de mots, supposé rationnel. Il existe donc un automate de mots déterministe $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ reconnaissant L. Soient $q'_1, q'_2 \notin Q$. On construit l'automate d'arbres descendant déterministe $\mathcal{A}^{\downarrow} = (Q \cup \{q'_1, q'_2\}, q_0, F \cup \{q'_1\}, \delta')$ avec pour $(q, \alpha) \in Q \times \Sigma, \delta'(q, \alpha) \coloneqq (\delta(q, \alpha), q'_1)$ et, pour $q \in \{q'_1, q'_2\}$ et pour $\alpha \in \Sigma$, $\delta'(q, \alpha) \coloneqq (q'_2, q'_2)$. Démontrer que $\mathcal{A}^{\downarrow}$ reconnaît $\mathit{chaîne}(L)$.

☐ 15 – Montrer que pour tout langage de mots L, si $\mathit{chaîne}(L)$ est reconnu par un automate d'arbres descendant déterministe, alors L est rationnel.

☐ 16 – Soit $L_{\text{égal}}$ le langage de mots sur l'alphabet $\{\alpha_0, \alpha_1\}$ formé des mots contenant autant de α_0 que de α_1. Supposons par l'absurde qu'il existe un automate (de mots) déterministe $\mathcal{A}_{\text{égal}}$ reconnaissant $L_{\text{égal}}$ et soit k le nombre d'états de $\mathcal{A}_{\text{égal}}$. En considérant le mot $x = \alpha_0^k \alpha_1^k$, montrer que l'on aboutit à une contradiction, et que donc $L_{\text{égal}}$ n'est pas un langage rationnel.

En déduire qu'il n'existe aucun automate descendant déterministe reconnaissant $\mathit{chaîne}(L_{\text{égal}})$.

Automates d'arbres ascendants

Un *automate d'arbres ascendant* (ou simplement *automate ascendant*) sur l'alphabet Σ est un quadruplet $\mathcal{A}^{\uparrow} = (Q, I, F, \Delta)$ où :

(i) Q est un ensemble fini non vide dont les éléments sont appelés *états* ;
(ii) $I \subseteq Q$ est un ensemble dont les éléments sont appelés *états initiaux* ;
(iii) $F \subseteq Q$ est un ensemble dont les éléments sont appelés *états finals* ;
(iv) $\Delta : Q \times Q \times \Sigma \to \mathcal{P}(Q)$, où $\mathcal{P}(X)$ désigne l'ensemble des parties de X, est une application appelée *fonction de transition* ; pour tout $(q_g, q_d) \in Q \times Q$, et tout $\alpha \in \Sigma$, $\Delta(q_g, q_d, \alpha)$ est un ensemble d'états.

Par exemple, on définit un automate ascendant $\mathcal{A}_0^{\uparrow} = (Q, I, F, \Delta)$ sur l'alphabet $\{\alpha_0, \alpha_1\}$ avec :

(i) $Q = \{q_0, q_1\}$;
(ii) $I = \{q_0\}$;
(iii) $F = \{q_1\}$;
(iv) Δ est donnée par la table de transition suivante :

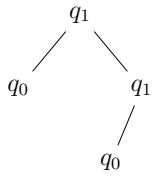
	Σ	
$Q \times Q$	α_0	α_1
(q_0, q_0)	$\{q_1\}$	$\{q_0\}$
(q_0, q_1)	$\{q_1\}$	$\{q_1\}$
(q_1, q_0)	$\{q_1\}$	$\{q_1\}$
(q_1, q_1)	$\{q_1\}$	$\{q_1\}$

Un automate ascendant $\mathcal{A}^{\uparrow} = (Q, I, F, \Delta)$ *reconnaît* un arbre t si :

— soit $t = \varepsilon$ et $I \cap F \neq \varnothing$;
— soit $t = (S, r, \lambda, g, d)$ et il existe une application $\varphi : S \to Q$ avec :
  (i) $\varphi(r) \in F$;
  (ii) pour tout $u \in S$, il existe $(q_g, q_d) \in Q \times Q$ tels que $\varphi(u) \in \Delta(q_g, q_d, \lambda(u))$ et :
    — si $g(u)$ est défini ($u \in S_g$), alors on a $\varphi(g(u)) = q_g$, sinon $q_g \in I$;
    — si $d(u)$ est défini ($u \in S_d$), alors on a $\varphi(d(u)) = q_d$, sinon $q_d \in I$.

Noter que, contrairement au cas des automates descendants déterministes, quand une telle application φ existe, elle n'est pas nécessairement unique.

On observe que $\mathcal{A}_0^{\uparrow}$ ne reconnaît pas ε (car $\{q_0\} \cap \{q_1\} = \varnothing$) et que $\mathcal{A}_0^{\uparrow}$ reconnaît l'arbre t_0 défini page 3 *via* l'application représentée ci-dessous :



Le *langage reconnu* par un automate ascendant $\mathcal{A}^{\uparrow}$, noté $\mathcal{L}(\mathcal{A}^{\uparrow})$, est l'ensemble de tous les arbres reconnus par $\mathcal{A}^{\uparrow}$. On dit qu'un langage d'arbres L est *rationnel* s'il existe un automate ascendant $\mathcal{A}^{\uparrow}$ qui reconnaît L. [1]

On dit qu'un automate ascendant (Q, I, F, Δ) est *déterministe* si $|I| = 1$ et, pour tout $(q_g, q_d, \alpha) \in Q \times Q \times \Sigma$, $|\Delta(q_g, q_d, \alpha)| = 1$.

☐ 17 – Montrer que l'on a $\mathcal{L}(\mathcal{A}_0^{\uparrow}) = L_0$.

☐ 18 – Soit L un langage d'arbres. Montrer que s'il existe un automate descendant déterministe $\mathcal{A}^{\downarrow}$ reconnaissant L, alors L est un langage d'arbres rationnel.

En Caml, un état q_i de $Q = \{q_0, \ldots, q_{n-1}\}$ est codé par l'entier i. L'ensemble des états initiaux I est codé par leur liste `initiaux_asc`, dans un ordre arbitraire ; l'ensemble des états finals F est codé par un vecteur de booléens `finals_asc` de taille n, dont la composante de position i contient *Vrai* si et seulement si $q_i \in F$. Finalement, les transitions sont codées par un tableau tridimensionnel de listes d'entiers, telle que `transitions_asc.(i).(j).(k)` est une liste dans un ordre arbitraire des états `q` avec $q \in \Delta(q_i, q_j, \alpha_k)$.

On représente ainsi en Caml un automate ascendant (Q, I, F, Δ) avec le type ci-dessous :

```
type Automate_Ascendant = {
  initiaux_asc: int list;
  finals_asc: bool vect;
  transitions_asc: int list vect vect vect
};;
```

1. La notion de langages d'arbres rationnels est distincte de la notion de langages de mots rationnels.

L'automate $\mathcal{A}_0^{\uparrow}$ peut alors être codé par :

```
let aa0 = {
  initiaux_asc=[0];
  finals_asc = [| false; true |];
  transitions_asc=[|
    [| [| [1]; [0] |]; [| [1]; [1] |] |];
    [| [| [1]; [1] |]; [| [1]; [1] |] |]
  |]
};;
```

☐ 19 – Coder une fonction `nombre_etats_asc` prenant en argument la représentation `aa` d'un automate ascendant et renvoyant le nombre d'états de cet automate.

☐ 20 – Coder une fonction `nombre_symboles_asc` prenant en argument la représentation `aa` d'un automate ascendant et renvoyant le nombre de symboles de l'alphabet sur lequel cet automate est défini.

☐ 21 – Coder une fonction Caml `applique_asc` telle que `applique_asc aa t`, où `aa` représente un automate ascendant $\mathcal{A}^{\uparrow} = (Q, I, F, \Delta)$ et `t` un arbre t, renvoie une liste sans doublon des états q pour lesquels il existe une application $\varphi : S \to Q$ avec $\varphi(r) = q$ qui vérifie la condition (ii) de la définition de reconnaissance d'un arbre par un automate ascendant page 7. Si $t = \varepsilon$, la fonction `applique_asc` doit renvoyer la liste des états initiaux de $\mathcal{A}^{\uparrow}$. On pourra utiliser les fonctions utilitaires des questions 1 à 4.

☐ 22 – En utilisant `applique_asc`, coder une fonction `evalue_asc` telle que `evalue_asc aa t`, où `aa` représente un automate ascendant $\mathcal{A}^{\uparrow}$ et `t` un arbre t, renvoie un booléen qui vaut *Vrai* si et seulement si $\mathcal{A}^{\uparrow}$ reconnaît t. On pourra utiliser la fonction `contient`.

☐ 23 – Montrer qu'un langage d'arbres L est un langage d'arbres rationnel si et seulement s'il existe un automate ascendant *déterministe* reconnaissant L.

☐ 24 – Coder deux fonctions Caml `identifiant_partie: int list -> int` et `partie_identifiant: int -> int list` réciproques l'une de l'autre, codant une bijection entre les parties de $[\![0\,;n-1]\!]$ (une partie étant représentée par une liste d'entiers sans doublon, dans un ordre arbitraire) et les entiers de 0 à $2^n - 1$. On rappelle qu'en Caml l'expression `1 lsl i` calcule l'entier 2^i.

☐ 25 – En s'appuyant sur `identifiant_partie` et `partie_identifiant` et sur la réponse à la question 23, coder une fonction `determinise_asc` prenant en argument la représentation `aa` d'un automate ascendant $\mathcal{A}^{\uparrow}$ et renvoyant la représentation d'un automate ascendant déterministe reconnaissant le même langage que $\mathcal{A}^{\uparrow}$.

☐ 26 – Montrer que si L est un langage d'arbres rationnel, alors $\mathcal{T}^{\Sigma}\backslash L$ est un langage d'arbres rationnel.

☐ 27 – Coder une fonction `complementaire_asc` prenant en entrée la représentation `aa` d'un automate ascendant reconnaissant un langage L et renvoyant la représentation d'un automate ascendant reconnaissant $\mathcal{T}^{\Sigma}\backslash L$.

☐ 28 – Montrer que si L_1 et L_2 sont deux langages d'arbres rationnels, alors $L_1 \cup L_2$ est un langage d'arbres rationnel.

☐ 29 – Coder une fonction `union_asc` prenant en entrée les représentations `aa1` et `aa2` de deux automates ascendants reconnaissant respectivement les langages L_1 et L_2 et renvoyant la représentation d'un automate ascendant reconnaissant $L_1 \cup L_2$.

☐ 30 – Montrer que si L_1 et L_2 sont deux langages d'arbres rationnels, alors $L_1 \cap L_2$ est un langage d'arbres rationnel.

☐ 31 – Coder une fonction `intersection_asc` prenant en entrée les représentations `aa1` et `aa2` de deux automates ascendants reconnaissant respectivement les langages L_1 et L_2 et renvoyant la représentation d'un automate ascendant reconnaissant $L_1 \cap L_2$.

☐ 32 – Sans chercher à utiliser les propriétés de clôture par union, complémentation ou intersection, montrer que le langage $L_{\text{impartial}}$ n'est pas un langage d'arbres rationnel.

Fin de l'épreuve