DS 4 : Option info MP/MP*

Devoir blanc

⋆ ⋆ ⋆

Consignes à lire très attentivement !

Durée de l'épreuve : 3 heures.

À la fin des 3 heures, merci de rendre votre copie et de quitter la salle **en silence** afin de ne pas déranger vos camarades qui continuent à composer.

On propose deux sujets distincts pour ce devoir. Vous devez en **choisir un** et n'en traiter **qu'un seul**. Indiquer **clairement** en début de copie le sujet choisi. Les deux sujets sont :

- **Type CCINP**. Le premier sujet est un sujet CCINP. Il commence à la page suivante et fait 12 pages. Les pages sont numérotées de 2/22 à 13/22. Les trois parties qui composent ce sujet sont indépendantes et peuvent être traitées par dans un ordre quelconque.
- **Type Mines-Ponts**. Le deuxième sujet est un sujet Mines-Ponts. Il commence à la 14ème page de ce document. Les pages sont numérotées de 2/10 à 10/10. Ce sujet est composé d'un unique problème, comportant 32 questions.

⋆ ⋆ ⋆

Partie I : logique et calcul des propositions

Dans une association de jeunes détectives, les membres s'entraînent à résoudre des problèmes logiques. Ceux-ci respectent les règles suivantes : « Lors d'une conversation, un même membre aura un comportement constant : il dira toujours la vérité, ou ne dira jamais la vérité. » et « Une conversation ne doit pas être absurde. »

Question I.1 *Soient n membres intervenant dans une même conversation. Chaque membre est représenté par une variable propositionnelle M_i avec $i \in [1 \cdots n]$ qui représente le fait que ce membre dit, ou ne dit pas, la vérité. Chaque membre fait une seule déclaration représentée par la variable propositionnelle D_i. Représenter le respect des règles dans cette conversation sous la forme d'une formule du calcul des propositions dépendant des variables M_i et D_i avec $i \in [1 \cdots n]$.*

Vous assistez à deux conversations sur la véracité des déclarations de deux groupes de trois membres de cette association.
Nous nommerons A, B et C les participants de la première conversation.

A : « Les seuls qui disent la vérité ici sont C et moi. »

B : « C ne dit pas la vérité. »

C : « Soit B dit la vérité. Soit A ne dit pas la vérité. »

Nous noterons A, B et C les variables propositionnelles associées au fait que A, B et C disent respectivement la vérité.
Nous noterons D_A, D_B et D_C les formules du calcul des propositions associées respectivement aux déclarations de A, B et C dans la conversation.

Question I.2 *Représenter les déclarations de la première conversation sous la forme de formules du calcul des propositions D_A, D_B et D_C dépendant des variables A, B et C.*

Question I.3 *En utilisant le calcul des propositions (résolution avec les tables de vérité), déterminer si A, B et C disent, ou ne disent pas, la vérité.*

Nous nommerons D, E et F les participants de la seconde conversation.

D : « Personne ne doit croire F. »

E : « D et F disent toujours la vérité. »

F : « E a dit la vérité. »

Nous noterons D, E et F les variables propositionnelles associées au fait que D, E et F disent respectivement la vérité.
Nous noterons D_D, D_E et D_F les formules du calcul des propositions associées respectivement aux déclarations de D, E et F dans la seconde conversation.

Question I.4 *Représenter les déclarations de la seconde conversation sous la forme de formules du calcul des propositions D_D, D_E et D_F dépendant des variables D, E et F.*

Question I.5 *En utilisant le calcul des propositions (résolution avec les formules de De Morgan), déterminer si D, E et F disent, ou ne disent pas, la vérité.*

Partie II : automates et langages

Le but de cet exercice est de prouver qu'un homomorphisme de langage préserve la structure de langage régulier : l'image et l'image réciproque d'un langage régulier par un homomorphisme de langage sont des langages réguliers. Pour simplifier les preuves, nous nous limiterons à des homomorphismes Λ-libres. Les résultats étudiés s'étendent au cas des homomorphismes quelconques.

1 Homomorphisme de langage

Déf. II.1 (Alphabets, mots, langages) Un alphabet X est un ensemble de symboles. Un mot $x_1 \dots x_n$ de taille n sur X est une séquence de taille n de symboles $x_i \in X$. ε est le symbole représentant le mot vide ($\varepsilon \notin X$). $X^\star$ est l'ensemble contenant ε et tous les mots sur X. L'opérateur binaire $\cdot$ est tel que : $\forall x \in X, \forall m \in X^\star, x \cdot m \in X^\star$. Un mot $x_1 \dots x_n$ de taille n s'écrit de manière unique $x_1 \cdot (x_2 \cdot (\dots \cdot (x_n \cdot \varepsilon) \dots))$. Un langage sur X est un sous-ensemble de $X^\star$. L'opérateur binaire associatif sur $X^\star$ de concaténation des mots $\centerdot$ admet ε comme élément neutre.

Déf. II.2 (Homomorphisme de langage) Soient deux alphabets X et Y, un homomorphisme de langage h est une application de domaine $X^\star$ et co-domaine $Y^\star$ qui préserve le mot vide et l'opérateur de concaténation des mots, c'est-à-dire :

(note : le co-domaine est l'ensemble d'arrivée)

$$h(\varepsilon) = \varepsilon$$
$$\forall m_1, m_2 \in X^\star, h(m_1 \centerdot m_2) = h(m_1) \centerdot h(m_2)$$

Déf. II.3 (Homomorphisme ε-libre) Soit un alphabet X, soit h un homomorphisme de langage de domaine $X^\star$, h est ε-libre, si et seulement si :

$$\forall m \in X^\star, m \neq \varepsilon \implies h(m) \neq \varepsilon$$

Déf. II.4 (Image d'un langage par un homomorphisme) Soit un alphabet X, soit un homomorphisme de langage h de domaine $X^\star$, soit $L_X \subseteq X^\star$ un langage sur X, l'image de L_X par h est définie par :

$$h(L_X) = \{h(m) \mid m \in L_X\}$$

Déf. II.5 (Image réciproque d'un langage par un homomorphisme) Soit un alphabet Y, soit un homomorphisme de langage h de co-domaine $Y^\star$, soit $L_Y \subseteq Y^\star$ un langage sur Y, l'image réciproque de L_Y par h est définie par :

$$h^{-1}(L_Y) = \{m \mid h(m) \in L_Y\}$$

Nous décomposons l'étude de la préservation de la structure de langage régulier par les homomorphismes de langage en plusieurs étapes qui reposent sur une restriction des homomorphismes de langage à un alphabet.

Déf. II.6 (Extension de Kleene) Soient deux alphabets X et Y, soit une application h de domaine X et co-domaine $Y^\star$, nous notons $\widehat{h}$ l'extension au domaine $X^\star$ définie par :

$$\widehat{h}(\varepsilon) = \varepsilon$$
$$\forall x \in X, m \in X^\star, \widehat{h}(x \cdot m) = h(x) \centerdot \widehat{h}(m)$$

Question II.1 *Soient deux alphabets X et Y, soit une application h de domaine X et co-domaine $Y^\star \setminus \{\varepsilon\}$, montrer que $\widehat{h}$ est un homomorphisme de langage ε-libre.*

Question II.2 *Soient deux alphabets X et Y, soit un homomorphisme de langage h de domaine $X^\star$ et co-domaine $Y^\star$, soit $h_{|X}$ sa restriction au domaine X, montrer que $\widehat{h_{|X}} = h$.*

2 Langages et expressions régulières

Pour montrer que l'image d'un langage régulier par un homomorphisme de langage est un langage régulier, nous exploitons la définition des langages réguliers sous la forme d'expressions régulières.

Déf. II.7 (Expression régulière) Soit un alphabet X, une expression régulière sur X est construite à partir des constantes $\emptyset$, ε et $a \in X$, de l'opérateur unaire de répétition $\star$ et des opérateurs binaires associatifs de concaténation $\cdot$ et de choix $\mid$ (qui est également commutatif). Les opérateurs ont les priorités croissantes suivantes : $\mid$, $\cdot$ et $\star$.

Exemple II.1 (Expression régulière) Soit l'alphabet $X = \{a, b\}$, l'expression suivante est une expression régulière sur X : $a \cdot (b \cdot a \cdot \varepsilon \mid a \cdot b \cdot a \cdot \varepsilon)^\star$. Comme indiqué dans la définition des mots, pour simplifier l'écriture, nous n'utilisons pas explicitement l'opérateur $\cdot$ et ε. L'expression précédente sera donc notée $a(ba \mid aba)^\star$.

Déf. II.8 (Langage spécifié par une expression régulière) Soit l'alphabet X, soit e une expression régulière sur X, le langage régulier $L(e) \subseteq X^\star$ spécifié par e est défini par :

$$
\begin{array}{l}
L(\emptyset) = \{\} \\
L(\varepsilon) = \{\varepsilon\} \\
L(a) = \{a\} \text{ si } a \in X \\
L(e_1 \mid e_2) = L(e_1) \cup L(e_2) \\
L(e_1 \cdot e_2) = \{m_1 \cdot m_2, m_1 \in L(e_1), m_2 \in L(e_2)\} \\
L(e^\star) = L(e)^\star
\end{array}
$$

Déf. II.9 (Homomorphisme d'expression régulière) Soient deux alphabets X et Y, soit h une application de domaine X et co-domaine $Y^\star$, soit e une expression régulière sur X, nous notons $\widetilde{h}$ l'application, dont le domaine est l'ensemble des expressions régulières sur X, définie par :

$$
\begin{array}{l}
\widetilde{h}(\emptyset) = \emptyset \\
\widetilde{h}(\varepsilon) = \varepsilon \\
\widetilde{h}(a) = h(a) \text{ si } a \in X \\
\widetilde{h}(e_1 \mid e_2) = \widetilde{h}(e_1) \mid \widetilde{h}(e_2) \\
\widetilde{h}(e_1 \cdot e_2) = \widetilde{h}(e_1) \cdot \widetilde{h}(e_2) \\
\widetilde{h}(e^\star) = \widetilde{h}(e)^\star
\end{array}
$$

Question II.3 *Soient les alphabets $X = \{a, b\}$ et $Y = \{0, 1\}$, soit l'application h de domaine X et co-domaine $Y^\star$ définie par $h(a) = 10$ et $h(b) = 0$, calculer $\widetilde{h}$ appliquée sur l'expression régulière de l'exemple II.1 (ci-dessus).*

Question II.4 *Soient deux alphabets X et Y, soit h une application de domaine X et co-domaine $Y^\star$, soit e une expression régulière sur X, montrer que $\widetilde{h}(e)$ est une expression régulière sur Y.*

Question II.5 *Soient deux alphabets X et Y, soit h une application de domaine X et co-domaine $Y^\star$, soit e une expression régulière sur X, montrer que $L(\widetilde{h}(e)) = \widehat{h}(L(e))$.*

Question II.6 *Soient deux alphabets X et Y, soit h un homomorphisme de langage ε-libre de domaine $X^\star$ et co-domaine $Y^\star$, soit L_X un langage régulier sur X, montrer que $h(L_X)$ est un langage régulier sur Y.*

3 Langages réguliers et automates finis

On rappelle que les langages réguliers sont exactement les langages reconnus par les automates finis. Pour montrer que l'image réciproque d'un langage régulier par un homomorphisme de langage est un langage régulier, nous exploitons la définition des langages réguliers sous la forme d'automates finis. Pour simplifier les preuves, nous nous limiterons au cas des automates finis non-déterministes (i sans transition arbitraire sur ε). Les résultats étudiés s'étendent au cadre des automates finis quelconques.

Déf. II.10 (Automate fini non-déterministe) Soit l'alphabet X, un automate fini non-déterministe sur X est un quintuplet $A = (Q, X, I, T, \delta)$ composé de :

- un ensemble fini d'états : Q ;
- un ensemble d'états initiaux : $I \subseteq Q$;
- un ensemble d'états terminaux : $T \subseteq Q$;
- une fonction de transition confondue avec son graphe : $\delta \subseteq Q \times X \mapsto \mathcal{P}(Q)$.

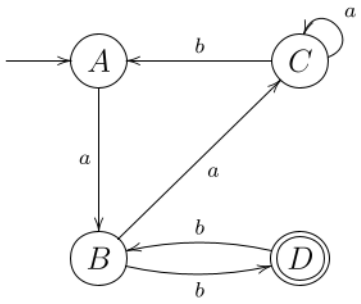
Pour une transition $\delta(o, x) = \{d_1, \ldots, d_n\}$ donnée, nous appelons o l'origine de la transition, x l'étiquette de la transition et d_1, ..., d_n les différentes destinations de la transition.
Les automates peuvent être représentés par un schéma suivant les conventions :

- les valeurs de la fonction de transition δ sont représentées par un graphe orienté dont les nœuds sont les états et les arêtes sont les transitions ;
- tout état q est entouré d'un cercle (q) ;
- tout état initial i est désigné par une flêche $\longrightarrow (i)$;
- tout état terminal t est entouré d'un second cercle $((t))$;
- toute arête étiquetée par le symbole $x \in X$ va de l'état o à l'état d si et seulement si $d \in \delta(o, x)$.

Exemple II.2 L'automate $\mathcal{E} = (Q, X, I, T, \delta)$ avec :

$$
\begin{array}{l}
Q = \{A, B, C, D\} \\
X = \{a, b\} \\
I = \{A\} \\
T = \{D\} \\
\quad \delta(A, a) = \{B\} \quad \delta(B, a) = \{C\} \\
\quad \delta(B, b) = \{D\} \quad \delta(C, a) = \{C\} \\
\quad \delta(C, b) = \{A\} \quad \delta(D, b) = \{B\}
\end{array}
$$

est représenté par le graphe suivant :



Déf. II.11 (Fermeture réflexive et transitive) Soit (Q, X, I, T, δ) un automate fini non-déterministe, $\delta^\star \subseteq Q \times X^\star \mapsto \mathcal{P}(Q)$ la fermeture réflexive et transitive de δ sur $X^\star$ est définie par :

$$\forall q \in Q,\ \delta^\star(q, \varepsilon) = \{q\}$$

$$\left\{ \begin{array}{l} \forall x \in X, \\ \forall m \in X^\star, \\ \forall o \in Q, \\ \forall d \in Q, \end{array} \right. \quad d \in \delta^\star(o, x \cdot m) \Leftrightarrow \exists q \in Q,\ (q \in \delta(o, x) \wedge d \in \delta^\star(q, m))$$

Déf. II.12 (Langage reconnu par un automate) Soit $\mathcal{A} = (Q, X, I, T, \delta)$ un automate fini non-déterministe, $L(\mathcal{A})$ le langage régulier sur X reconnu par $\mathcal{A}$ est défini par :

$$L(\mathcal{A}) = \{m \in X^\star \mid \exists i \in I, \exists t \in T,\ t \in \delta^\star(i, m)\}$$

Question II.7 *Donner, sans justification, l'expression régulière ou ensembliste représentant le langage sur $X = \{a, b\}$ reconnu par l'automate $\mathcal{E}$ de l'exemple II.2 (page 5).*

Question II.8 *Soit (Q, X, I, T, δ) un automate fini non-déterministe, montrer que :*

$$\forall o, d \in Q, \forall m_1, m_2 \in X^\star, d \in \delta^\star(o, m_1 \centerdot m_2) \Leftrightarrow \exists q \in Q, (q \in \delta^\star(o, m_1) \wedge d \in \delta^\star(q, m_2)$$

L'image réciproque du langage reconnu par un automate fini non-déterministe par un homomorphisme de langage peut être obtenue par transformation de cet automate selon la définition suivante.

Déf. II.13 (Image réciproque d'un automate fini) Soient X et Y deux alphabets, soit h une application de X dans $Y^\star$, soit $\mathcal{A} = (Q, Y, I, T, \delta)$ un automate fini non-déterministe, l'automate $\mathcal{B} = \widehat{h}^{-1}(\mathcal{A})$, image réciproque de l'automate $\mathcal{A}$, est défini par :

$$\mathcal{B} = (Q, X, I, T, \delta_{\widehat{h}^{-1}})$$
$$\forall x \in X,\ \forall o \in Q,\ \forall d \in Q,\ d \in \delta_{\widehat{h}^{-1}}(o, x) \Leftrightarrow o \in \delta^\star(d, \widehat{h}(x))$$

Question II.9 *Soient deux alphabets $X = \{a, b\}$ et $Y = \{0, 1\}$ soit h une application de domaine Y et co-domaine $X^\star$ telle que $h(0) = ab$ et $h(1) = b$, construire l'automate $\widehat{h}^{-1}(\mathcal{E})$ pour l'automate $\mathcal{E}$ de l'exemple II.2.*

Question II.10 *Caractériser le langage reconnu par $\widehat{h}^{-1}(\mathcal{E})$ par une expression régulière ou ensembliste. Comparer le langage reconnu par $\widehat{h}(\widehat{h}^{-1}(\mathcal{E}))$ avec le langage reconnu par $\mathcal{E}$.*

Question II.11 *Soit (Q, X, I, T, δ) un automate fini non-déterministe, soit une application h de domaine X et co-domaine $Y^\star$, montrer que :*

$$\forall m \in X^\star,\ \forall o \in Q,\ \forall d \in Q,\ d \in \delta^\star_{\widehat{h}^{-1}}(o, m) \Leftrightarrow d \in \delta^\star(o, \widehat{h}(m))$$

Question II.12 *Soit un automate fini non-déterministe $\mathcal{A}$, soit une application h de domaine X et co-domaine $Y^\star$, quelle relation liant les langages reconnus par $\mathcal{A}$ et $\widehat{h}^{-1}(\mathcal{A})$ peut-on déduire des questions précédentes ?*

Question II.13 *Soient deux alphabets X et Y, soit h un homomorphisme de langage ε-libre de domaine $X^\star$ et co-domaine $Y^\star$, soit L_Y un langage régulier sur Y, montrer que $h^{-1}(L_Y)$ est un langage régulier sur X.*

Partie III : algorithmique et programmation en CaML

Cette partie doit être traitée par les étudiants qui ont utilisé le langage CaML dans le cadre des enseignements d'informatique. Les fonctions écrites devront être récursives ou faire appel à des fonctions auxiliaires récursives. Elles ne devront pas utiliser d'instructions itératives (c'est-à-dire `for`, `while`, ...) ni de références.

1 Exercice : le tri par sélection

L'objectif de cet exercice est d'étudier une implantation particulière d'un algorithme de tri d'une séquence d'entiers.

Déf. III.1 Une séquence s de taille n de valeurs v_i avec $1 \leq i \leq n$ est notée $\langle v_n, \ldots, v_1 \rangle$. Sa taille n est notée $\mid s \mid$. Une même valeur v peut figurer plusieurs fois dans s. Nous noterons $card(v, s)$ le cardinal de v dans s, c'est-à-dire le nombre de fois que v figure dans s, défini par :

$$card(v, \langle v_n, \ldots, v_1 \rangle) = card\{i \in \mathbb{N} \mid 1 \leq i \leq n, v = v_i\}.$$

Le type `entiers` représente une séquence d'entiers. Le type `couple` représente un couple dont le premier élément est un entier et le second est une séquence d'entiers. Les définitions de ces types sont :

```
type entiers =  int list;;

type couple =  int * entiers;;
```

Soit le programme en langage CaML :

```
let selection s =
  let rec aux c a =
    match a with
      | [] -> (c,a)
      | t::q ->
        if (c < t) then
          let (m,r) = aux c q in
          (m,(t::r))
        else
          let (m,r) = aux t q in
          (m,(c::r))
  in
  aux (hd s) (tl s);;

let rec tri s =
  match s with
    | [] -> []
    | t::q ->
      let (m,r) = selection s in
      m :: (tri r);;
```

Soit la constante `exemple` définie et initialisée par :

```
let exemple = [ 3; 1; 4; 2];;
```

Question III.1 *Détailler les étapes du calcul de* `(trier exemple)` *en précisant pour chaque appel aux fonctions* `selection`, `aux` *et* `trier`, *la valeur du paramètre et du résultat.*

Déf. III.2 (Symbole de Kronecker) Soient deux valeurs v_1 et v_2, le symbole de Kronecker δ est une fonction $\delta(v_1, v_2)$ égale à la valeur 1 si v_1 et v_2 sont égales et à la valeur 0 sinon.

Question III.2 *Soit l'entier* `m`, *soient les séquences d'entiers* `s` $= \langle s_n, \ldots, s_1 \rangle$ *de taille* n *et* `r` $= \langle r_p, \ldots, r_1 \rangle$ *de taille* p, *tels que* `(m,r) = (selection s)`, *montrer que :*

(a) $\forall i, 1 \leq i \leq n, \delta(s_i, \mathtt{m}) + card(s_i, \mathtt{r}) = card(s_i, \mathtt{s})$

(b) $n = p + 1$

(c) $\forall i, 1 \leq i \leq p, \mathtt{m} \leq r_i$

Dans ce but, vous pouvez spécifier les propriétés que doit satisfaire la fonction `aux`. *Montrer que celles-ci sont satisfaites et les exploiter ensuite.*

Question III.3 *Soit la séquence* `s` $= \langle s_m, \ldots, s_1 \rangle$ *de taille* m, *soit la séquence* `r` $= \langle r_n, \ldots, r_1 \rangle$ *de taille* n, *telles que* `r = (trier s)`, *montrer que :*

(a) $m = n$

(b) $\forall i, 1 \leq i \leq m, card(s_i, \mathtt{s}) = card(s_i, \mathtt{r})$

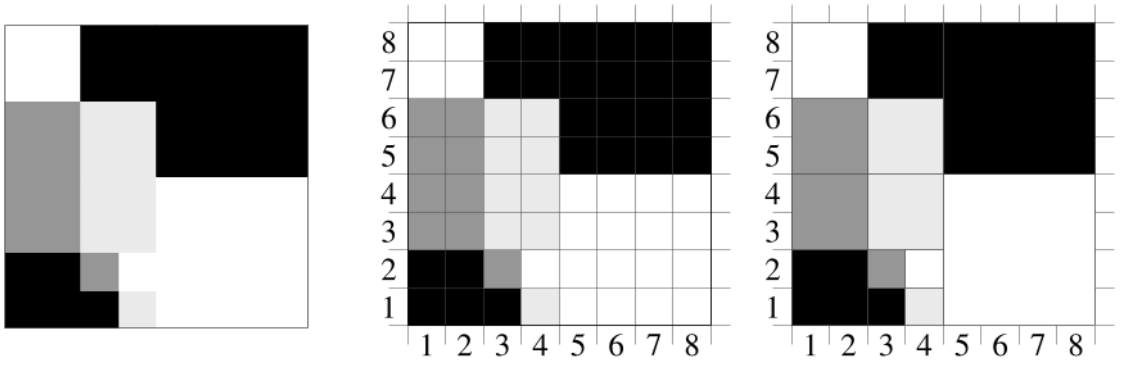
(c) $\forall i, 1 \leq i < m, r_{i+1} \leq r_i$

Question III.4 *Montrer que le calcul des fonctions* `selection`, `aux` *et* `trier` *se termine quelles que soient les valeurs de leurs paramètres respectant le type des fonctions.*

Question III.5 *Donner des exemples de valeurs du paramètre* `s` *de la fonction* `trier` *qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués.*
Montrer que la complexité de la fonction `trier` *en fonction du nombre* n *de valeurs dans les séquences données en paramètre est de* $O(\mid n \mid^2)$. *Cette estimation ne prend en compte que le nombre d'appels récursifs effectués.*

2 Problème : représentation d'images par des arbres quaternaires

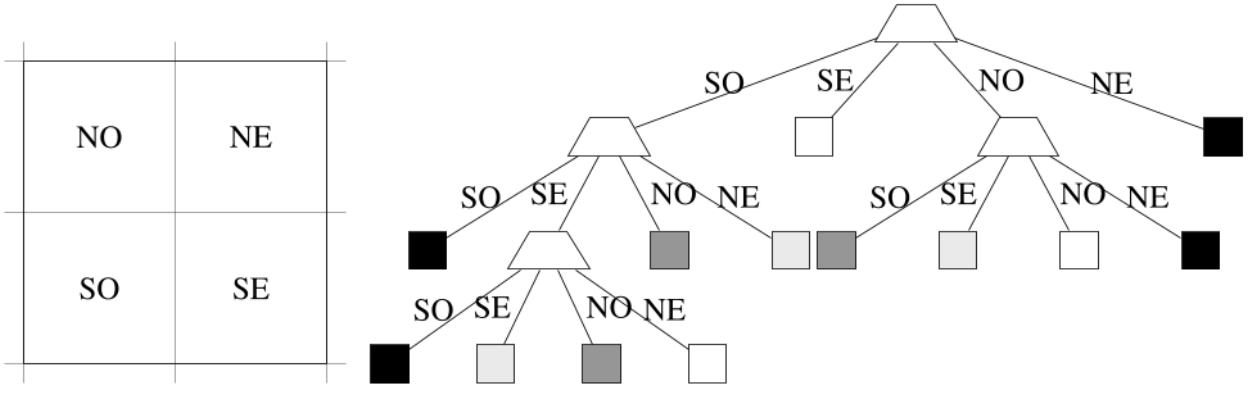
La structure d'arbre quaternaire est une extension de la structure d'arbre binaire qui permet de représenter des informations en deux dimensions. En particulier, la structure d'arbre binaire est utilisée pour représenter de manière plus compacte des images. Il s'agit de décomposer une image par dichotomie sur les deux dimensions jusqu'à obtenir des blocs de la même couleur.



(1-a) Brute (1-b) Par pixel (1-c) Par bloc

FIGURE 1: images

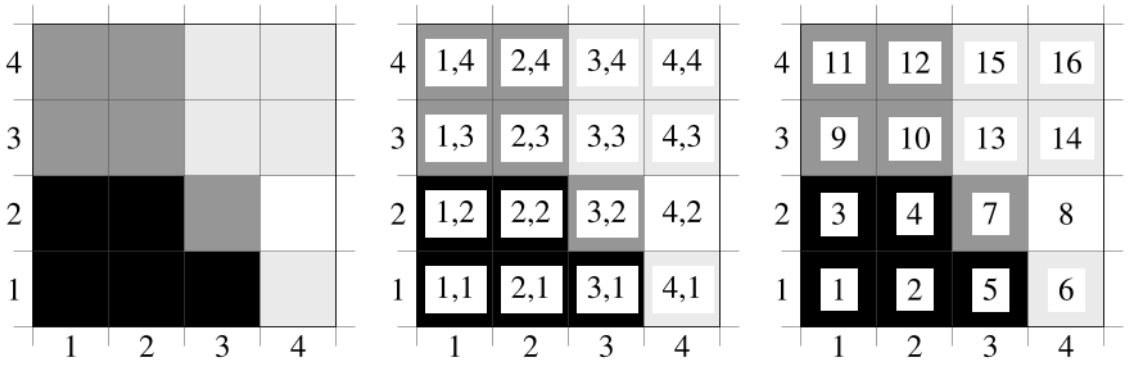
Les images des figures 1 et 2 illustrent ce principe. L'image brute (1-a) contient des carrés de différentes couleurs. Cette image est découpée uniformément en pixels (picture elements) dans l'image (1-b). Les pixels sont des carrés de la plus petite taille nécessaire. Ce découpage fait apparaître de nombreux pixels de la même couleur. Pour réduire la taille du codage, l'image (1-c) illustre un découpage dichotomique de l'image en carrés qui regroupent certains pixels de la même couleur. L'arbre quaternaire de l'image (2-b) représente les carrés contenus dans (1-c). Les étiquettes sur les fils de chaque nœud réprésentent la position géographique des fils dans le nœud : Sud-Ouest, Sud-Est, Nord-Ouest, Nord-Est comme indiqué dans (2-a).



(2-a) position géographique dans un nœud (2-b) codage de l'image de la figure 1 par un arbre quaternaire

FIGURE 2: arbre quaternaire

L'objectif de ce problème est l'étude de cette structure d'arbre quaternaire et de son utilisation pour le codage d'images en niveau de gris. Pour simplifier les programmes, nous nous limitons à des images carrées dont la longueur du côté est une puissance de 2. Les résultats étudiés se généralisent au cas des images rectangles de taille quelconque.



(3-a) Image extraite (3-b) Coordonnées (3-c) Numéros

FIGURE 3: identification des pixels

2.1 Arbres quaternaires

Déf. III.3 (Arbre quaternaire associé à une image) Un arbre quaternaire a qui représente une image carrée est composé de nœuds, qui peuvent être des blocs ou des divisions. Les ensembles des nœuds, des divisions et des blocs, de l'arbre a sont notés $\mathcal{N}(a)$, $\mathcal{D}(a)$ et $\mathcal{B}(a)$ avec $\mathcal{N}(a) = \mathcal{D}(a) \cup \mathcal{B}(a)$. Chaque nœud $n \in \mathcal{N}(a)$ contient une abscisse, une ordonnée et une taille, notées $\mathcal{X}(n)$, $\mathcal{Y}(n)$ et $\mathcal{T}(n)$, qui correspondent aux coordonnées et à la longueur du côté de la partie carrée de l'image représentée par n. Chaque bloc $b \in \mathcal{B}(a)$ contient une couleur notée $\mathcal{C}(n)$ qui correspond à la couleur de la partie carrée de l'image représentée par b. Chaque division $d \in \mathcal{D}(a)$ contient quatre fils $f_{SO}, f_{SE}, f_{NO}, f_{NE} \in \mathcal{N}(a)$. Ces fils sont indexés par la position relative de la partie carrée de l'image qu'ils représentent dans l'image représentée par la division d : Sud-Ouest, Sud-Est, Nord-Ouest et Nord-Est.

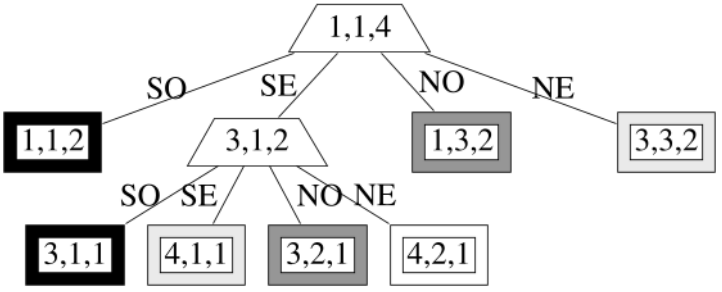


FIGURE 4: structure de données correspondant à l'image (3-a)

Exemple III.1 La figure 3 contient l'image (3-a) représentée par le sous-arbre situé au Sud-Ouest de l'arbre (2-b) page 9 ainsi que l'indication (3-b) des coordonnées de chaque point de cette image. Elle contient également la technique de numérotation des points exploitée dans la section 2.3 (page 13). La figure 4 (ci-dessus) contient l'arbre qui représente cette image dont les blocs et les divisions ont été annotés avec les coordonnées, tailles et couleurs.

Déf. III.4 (Profondeur d'un arbre quaternaire) La profondeur d'un bloc $b \in \mathcal{B}(a)$ d'un arbre quaternaire a est égale au nombre de divisions qui figurent dans la branche conduisant de la racine de a au bloc b. La profondeur d'un arbre a est le maximum des profondeurs de ses blocs $b \in \mathcal{B}(a)$.

Exemple III.2 La profondeur de l'arbre de la figure 2-b (page 9) est 3. La profondeur de l'arbre de la figure 4 (ci-dessus) est 2.

Déf. III.5 (Arbre quaternaire valide) Un arbre quaternaire a est valide, si et seulement si :

- les tailles, abscisses et ordonnées de chaque nœud $n \in \mathcal{N}(a)$ sont strictement positives ;
- les tailles des quatre fils $f_{SO}, f_{SE}, f_{NO}, f_{NE}$ de chaque division $d \in \mathcal{D}(a)$ sont identiques et égales à la moitié de la taille de d ;
- les abscisses et ordonnées des fils de chaque division $d \in \mathcal{D}(a)$ sont cohérentes avec la position géographique de chaque fils $f_{SO}, f_{SE}, f_{NO}, f_{NE}$ et avec l'abscisse, l'ordonnée et la taille de la division d ;
- au moins deux des quatre fils $f_{SO}, f_{SE}, f_{NO}, f_{NE}$ de chaque division $d \in \mathcal{D}(a)$ contiennent des blocs de couleurs différentes.

Question III.6 *Exprimer le fait qu'un arbre quaternaire a est valide sous la forme d'une propriété* $\mathrm{VAQ}(a)$.

2.2 Représentation en CaML

Un arbre quaternaire est représenté par le type `quater`. La position d'un sous-arbre dans un nœud est représentée par le type énuméré `position` contenant les valeurs `SO`, `SE`, `NO` et `NE` (représentant respectivement les sous-arbres situés au Sud-Ouest, Sud-Est, Nord-Ouest et Nord-Est d'une division). Les définitions en CaML de ces types sont :

```
type quater =
  | Division of int * int * int * quater * quater * quater * quater
  | Bloc of int * int * int * int;;
type position = SO | SE | NO | NE;;
```

Dans l'appel `Bloc(x,y,t,c)` qui construit un arbre quaternaire dont la racine est un bloc, les paramètres `x` et `y` sont les coordonnées du point situé en bas à gauche de l'image représentée par ce bloc, `t` est la longueur du côté de l'image carrée représentée par ce bloc et `c` est la couleur des points de l'image représentée par ce bloc.
Dans l'appel `Division(x,y,t,so,se,no,ne)` qui construit un arbre quaternaire dont la racine est une division, les paramètres `x` et `y` sont les coordonnées du point situé en bas à gauche de l'image représentée par cette division, `t` est la longueur du côté de l'image carrée représentée par cette division, `so`, `se`, `no` et `ne` sont quatre arbres quaternaires qui sont les quatre parties de la sub-division de l'image représentée par cette division. Celles-ci sont respectivement, `so` la partie en bas à gauche (Sud-Ouest), `se` la partie en bas à droite (Sud-Est), `no` la partie en haut à gauche (Nord-Ouest), `ne` la partie en haut à droite (Nord-Est).

Exemple III.3 En associant les valeurs `0`, `1`, `2`, `3` aux couleurs noir, gris foncé, gris clair et blanc, l'expression suivante :

```
let b11_2 = Bloc( 1, 1, 2, 0) in (* SO racine *)
let b31_1 = Bloc( 3, 1, 1, 0) in (* SO du SE racine *)
let b41_1 = Bloc( 4, 1, 1, 2) in (* SE du SE racine *)
let b32_1 = Bloc( 3, 2, 1, 1) in (* NO du SE racine *)
let b42_1 = Bloc( 4, 2, 1, 3) in (* NE du SE racine *)
let d31_2 = (* SE racine *)
  Division( 3, 1, 2, b31_1, b41_1, b32_1, b42_1) in
let b13_2 = Bloc( 1, 3, 2, 1) in (* NO racine *)
let b33_2 = Bloc( 3, 3, 2, 2) in (* NE racine *)
  Division( 1, 1, 4, b11_2, d31_2, b13_2, b33_2) (* racine *)
```

est alors associée à l'arbre quaternaire représenté graphiquement sur la figure 4 (page 10).

2.2.1 Scission d'un arbre quaternaire

Question III.7 *Ecrire en CaML une fonction* `scinder` *de type* `quater -> quater` *telle que l'appel* `(scinder a)` *sur un arbre quaternaire valide* `a` *renvoie, soit un arbre quaternaire identique à l'arbre* `a` *si la racine de celui-ci est une division, soit un arbre quaternaire dont la racine est une division contenant quatre blocs de même couleur identique à celle du bloc à la racine de l'arbre* `a`*. Les coordonnées et les tailles des blocs et de la division doivent être cohérentes. Toutes les contraintes de validité de l'arbre renvoyé doivent être satisfaites sauf celle concernant les couleurs.*

2.2.2 Fusion d'arbres quaternaires

Question III.8 *Ecrire en CaML une fonction* `fusionner` *de type* `quater -> quater -> quater -> quater -> quater` *telle que l'appel* `(fusionner so se no ne)` *sur quatre arbres quaternaires valides* `so`, `se`, `no` *et* `ne` *renvoie un arbre quaternaire valide codant l'image dont les parties Sud-Ouest, Sud-Est, Nord-Ouest et Nord-Est sont codées par* `so`, `se`, `no` *et* `ne`*. Les abscisses, ordonnées et tailles de* `so`, `se`, `no` *et* `ne` *sont cohérentes avec leur position dans l'image représentée par le résultat renvoyé.*

2.2.3 Calcul de la profondeur d'un arbre quaternaire

Question III.9 *Ecrire en CaML une fonction* `profondeur` *de type* `quater -> int` *telle que l'appel* `(profondeur a)` *sur un arbre quaternaire valide* `a` *renvoie la profondeur de l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

2.2.4 Consulter la couleur d'un point dans un arbre quaternaire

Question III.10 *Ecrire en CaML une fonction* `consulter` *de type* `int -> int -> quater -> int` *telle que l'appel* `(consulter x y a)` *sur un point d'abscisse* `x` *et d'ordonnée* `y` *et sur un arbre quaternaire valide* `a` *tel que le point d'abscisse* `x` *et d'ordonnée* `y` *soit contenu dans l'image représentée par l'arbre* `a`*, renvoie la couleur du point d'abscisse* `x` *et d'ordonnée* `y` *dans l'image représentée par l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

2.2.5 Peindre un point dans un arbre quaternaire

Question III.11 *Ecrire en CaML une fonction* `peindre` *de type* `int -> int -> int -> quater -> quater` *telle que l'appel* `(peindre x y c a)` *sur un point d'abscisse* `x` *et d'ordonnée* `y`*, une couleur* `c` *et un arbre quaternaire valide* `a` *renvoie un arbre quaternaire valide. Le point de coordonnées* `x` *et* `y` *doit être contenu dans l'image représentée par l'arbre* `a`*. L'arbre renvoyé représente une image contenant les mêmes couleurs que l'image représentée par l'arbre* `a` *sauf pour le point de coordonnées* `x` *et* `y` *dont la couleur sera* `c`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

2.2.6 Validation d'un arbre quaternaire

Question III.12 *Ecrire en CaML une fonction* `valider` *de type* `quater -> bool` *telle que l'appel* `(valider a)` *sur un arbre quaternaire* `a` *renvoie la valeur* `true` *si l'arbre* `a` *est valide, c'est-à-dire si* VAQ(a) *et la valeur* `false` *sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

2.3 Sauvegarde et restauration

Pour sauvegarder dans un fichier les couleurs contenues dans un arbre quaternaire valide, celles-ci sont rangées dans une séquence triée selon la position des points dans l'arbre. L'ordre choisi permet de restaurer l'arbre efficacement. La figure (3-c) (page 10) indique l'ordre dans lequel les points doivent être sauvegardés. La séquence manipulée contiendra les couleurs et les numéros associés à la position de chaque couleur dans l'arbre.

2.3.1 Codage en CaML

Une séquence de positions et de couleurs est représentée par le type *sequence* dont la définition est :

```
type sequence =  (int * int) list;;
```

La position figure avant la couleur associée dans la séquence.

Exemple III.4 En associant les valeurs `0`, `1`, `2`, `3` aux couleurs noir, gris foncé, gris clair et blanc, la sauvegarde de l'image de la figure (3-c) (page 10) produit la séquence :

```
[(1,0); (2,0); (3,0); (4,0); (5,0); (6,2); (7,1); (8,3);
 (9,1); (10,1); (11,1); (12,1); (13,2); (14,2); (15,2); (16,2)]
```

2.3.2 Sauvegarde d'un arbre quaternaire

Question III.13 *Ecrire en CaML une fonction* `sauvegarder` *de type* `quater -> sequence` *telle que l'appel* `(sauvegarder a)` *sur un arbre quaternaire valide* `a` *renvoie une séquence triée contenant les mêmes couleurs à la même position que l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a` *et ne devra pas reparcourir la (ou les) séquence(s) créée(s) en dehors de leur création. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

2.3.3 Restauration d'un arbre quaternaire

Question III.14 *Ecrire en CaML une fonction* `restaurer` *de type* `sequence -> quater` *telle que l'appel* `(restaurer s)` *sur une séquence valide* `s` *renvoie un arbre quaternaire valide contenant exactement les points contenus dans la séquence* `s`*. L'algorithme utilisé ne devra parcourir qu'une seule fois la séquence* `s` *et ne devra pas reparcourir les arbres quaternaires créés en dehors de leur création. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

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 :

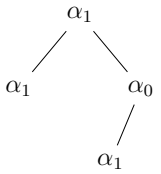
(i) S est un ensemble fini non vide dont les éléments sont appelés *nœuds* ;
(ii) $r \in S$ est la *racine* de S ;
(iii) $\lambda : S \to \Sigma$ est une application associant à chaque nœud de S une *étiquette* de Σ ;
(iv) $g : S_g \to S\backslash\{r\}$, où $S_g \subseteq S$, est une application injective 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 tableau 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 array;
  transitions_desc: (int*int) array array
};;
```

☐ 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 réguliers 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 **régulier**. 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 **régulier**.

☐ 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 régulier.

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 :

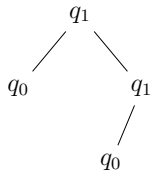
	Σ	
$Q \times Q$	α_0	α_1
(q_0, q_0)	$\{q_1\}$	$\{q_0\}$
(q_0, q_1)	$\{q_1\}$	$\{q_1\}$
(q_1, q_0)	$\{q_1\}$	$\{q_1\}$
(q_1, q_1)	$\{q_1\}$	$\{q_1\}$

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 tableau 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 array;
  transitions_asc: int list array array array
};;
```

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

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