Durée : 4h. Aucun matériel électronique autorisé. Aucun document autorisé.

Exercice 1 : Calcul des composantes connexes

Dans cet exercice, les graphes non orientés considérés sont de la forme $G = (S, A)$ où $S = [\![0, n[\![$ pour un certain $n \in \mathbb{N}$. On suppose alors qu'ils sont représentés par leur table de listes d'adjacence.

Q. 1 Donner le pseudo-code d'un algorithme permettant le calcul des composantes connexes d'un graphe non orienté. Cet algorithme devra retourner les composantes connexes sous la forme d'un tableau T indicé par $S = [\![0, n[\![$, contenant des entiers de l'intervalle $[\![0, p[\![$ où p est le nombre de composantes connexes du graphe. Cela signifie que pour tout $(u, v) \in [\![0, n[\![^2$, u et v sont dans la même composante connexe de G si et seulement si $T[i] = T[j]$.

Q. 2 Donner la complexité de l'algorithme proposé à la **Q. 1**.

Q. 3 L'algorithme proposé à la **Q. 1** permet-il aussi le calcul des composantes fortement connexes d'un graphe orienté ? Justifier ou donner un contre-exemple.

Q. 4 Donner un algorithme prenant en argument un graphe non orienté $G = (S, A)$ et une liste L de sommets de S, sans redondance, et retournant si oui ou non l'ensemble des sommets de L est connexe. Donner la complexité de cet algorithme.

Exercice 2 : Automates d'arbres [CCMP 2015]

Il est possible de faire appel à d'autres fonctions définies dans les questions précédentes ; et de 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 sans empattement (par exemple `n`).

1. Fonctions utilitaires

Dans cette partie, on code quelques fonctions sur les listes qui pourront être utiles par la suite.

Q. 1 Coder une fonction OCaml `contient : 'a list -> 'a -> bool` telle que `(contient li x)` renvoie un booléen qui vaut `true` si et seulement si la liste `li` contient l'élément `x`. Donner, sans justification, la complexité de cette fonction.

Q. 2 En utilisant la fonction `contient`, coder une fonction `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, sans justification, la complexité de cette fonction.

Q. 3 En utilisant la fonction `union`, coder une fonction OCaml `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`; `l_2`; …; `l_k`]` la liste codée par `l` et en posant $L = \sum_{j=1}^{k} |l_j|$, donner, sans justification, la complexité de la fonction `fusion` en fonction de L.

Q. 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, sans justification, la complexité de cette fonction.

2. Arbres binaires étiquetés

Alphabet. Soit $\Sigma = \{\alpha_0, \dots, \alpha_{m-1}\}$ un alphabet non vide de m symboles. En OCaml, on représentera le symbole α_k, pour $0 \leqslant k \leqslant m-1$, par l'entier k. Cet alphabet est supposé fixé dans tout l'énoncé.

Arbres binaires étiquetés par Σ. On considère dans cet exercice des arbres binaires étiquetés par des éléments de Σ, un tel arbre est :
- soit l'arbre vide noté E ;
- soit un nœud contenant une étiquette $x \in \Sigma$, un fils gauche g et un fils droit d qui sont eux-même des arbres étiquetés par Σ, on note un tel nœud $\mathsf{N}(x, g, d)$.

On note $\mathcal{T}^\Sigma$ l'ensemble des arbres étiquetés par Σ. De tels arbres seront représentés en OCaml au moyen du type ci-dessous.

```
type arbre =
  | E
  | N of int * arbre * arbre
```

Si $t \in \mathcal{T}^\Sigma$, on note $|t|$ la *taille* de t : $|\mathsf{E}| = 0$, $|\mathsf{N}(\_, g, d)| = 1 + |g| + |d|$.

Positions dans un arbre binaire. Les nœuds d'un arbre seront représentés par leur chemin depuis la racine de l'arbre, un tel chemin est un mot sur l'alphabet $\{\mathbf{0}, \mathbf{1}\}$: $\mathbf{0}$ est à comprendre comme un déplacement dans le fils gauche, $\mathbf{1}$ comme un déplacement dans le fils droit. Ainsi ε est le chemin menant à la racine et désigne donc la racine de l'arbre, $\mathbf{0}$ est le chemin menant de la racine au fils gauche de la racine (s'il existe) et désigne donc le fils gauche de la racine.

On définit, par induction sur $t \in \mathcal{T}^\Sigma$, l'ensemble des *positions des nœuds* de l'arbre t, noté $\mathcal{C}_N(t)$ par :
- $\mathcal{C}_N(\mathsf{E}) = \emptyset$;
- $\mathcal{C}_N(\mathsf{N}(x, g, d)) = \{\varepsilon\} \cup \{\mathbf{0}\}\cdot\mathcal{C}_N(g) \cup \{\mathbf{1}\}\cdot\mathcal{C}_N(d)$.

Afin d'identifier, non seulement les nœuds, mais aussi les sous-arbres vides d'un arbre, on définit l'ensemble de *toutes les positions* d'un arbre t, noté $\mathcal{C}(t)$ par :

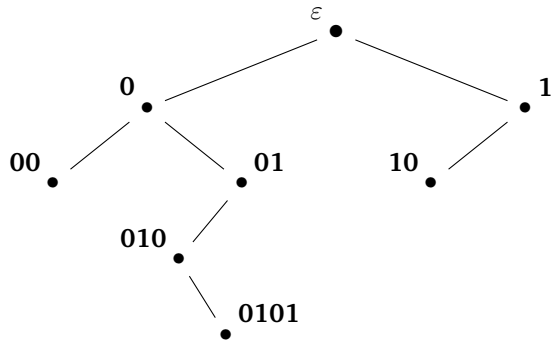
$$\mathcal{C}(t) = \{\varepsilon\} \cup \mathcal{C}_N(t) \cup \{w\cdot\{\mathbf{0}\} \mid w \in \mathcal{C}_N(t)\} \cup \{w\cdot\{\mathbf{1}\} \mid w \in \mathcal{C}_N(t)\}.$$

Finalement, on définit l'ensemble des *positions des extrémités*, noté $\mathcal{C}_E(t)$, par :

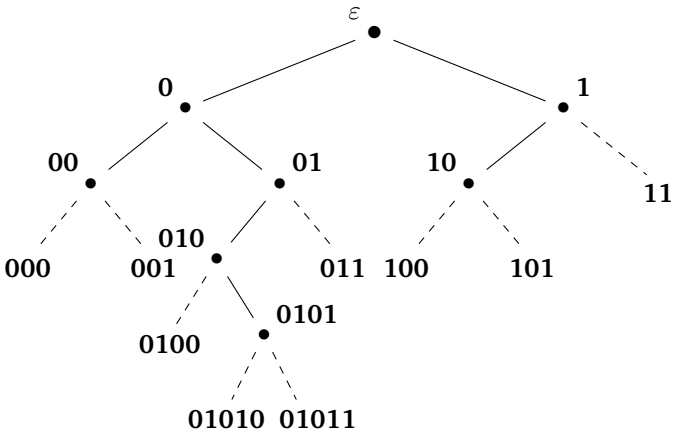
$$\mathcal{C}_E(t) = \mathcal{C}(t) \setminus \mathcal{C}_N(t).$$

Remarquons donc que l'arbre vide ne contient pas de nœud, ainsi $\mathcal{C}_N(\mathsf{E}) = \emptyset$, il admet cependant une extrémité : $\mathcal{C}(\mathsf{E}) = \{\varepsilon\}$ et donc $\mathcal{C}_E(\mathsf{E}) = \{\varepsilon\}$.

On illustre ci-dessous ces ensembles pour un arbre t_{ex} (dont les étiquettes ne sont pas représentées).



Positions des nœuds de l'arbre t_{ex}.



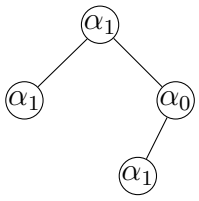
Toutes les positions de l'arbre t_{ex}.

$$\mathcal{C}_N(t_{\text{ex}}) = \{\varepsilon, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{01}, \mathbf{010}, \mathbf{0101}, \mathbf{10}\}$$
$$\mathcal{C}_E(t_{\text{ex}}) = \{\mathbf{000}, \mathbf{001}, \mathbf{0100}, \mathbf{01010}, \mathbf{01011}, \mathbf{011}, \mathbf{100}, \mathbf{101}, \mathbf{11}\}$$
$$\mathcal{C}(t_{\text{ex}}) = \{\varepsilon, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{000}, \mathbf{001}, \mathbf{01}, \mathbf{011}, \mathbf{010}, \mathbf{0101}, \mathbf{01010}, \mathbf{01011}, \mathbf{0100}, \mathbf{10}, \mathbf{11}, \mathbf{100}, \mathbf{101}\}$$

On définit inductivement le *sous-arbre* enraciné à une position $p \in \mathcal{C}_N(t)$ dans un arbre t, noté $t_{|p}$ par :

- $t_{|\varepsilon} = t$;
- $t_{|\mathbf{0}\cdot w} = g_{|w}$ si $t = \mathsf{N}(x, g, d)$;
- $t_{|\mathbf{1}\cdot w} = d_{|w}$ si $t = \mathsf{N}(x, g, d)$.

Exemple. Dans la suite, on considère l'arbre exemple t_0, représenté ci-contre, et défini par la valeur OCaml ci-dessous.




```
let t0 =
  N(1,
    N(1, E, E),
    N(0, N(1, E, E), E))
```

Q. 5 Donner les ensembles $\mathcal{C}_N(t_0)$ et $\mathcal{C}_E(t_0)$.

Q. 6 *a)* Donner et prouver par induction sur $t \in \mathcal{T}_\Sigma$, une relation entre le cardinal de $\mathcal{C}_N(t)$ et la taille de t.

b) Donner, sans la démontrer, une relation entre le cardinal de $\mathcal{C}_E(t)$ et la taille de t pour un arbre quelconque $t \in \mathcal{T}_\Sigma$.

Q. 7 On représente en OCaml, une position par une liste de booléens, par exemple **100** est représentée par `[true; false; false]`, ε est représentée par `[]`. Définir une fonction `pos_noeuds : arbre -> bool list list` prenant en argument un arbre t et retournant une liste sans doublons des éléments de $\mathcal{C}_N(t)$.

3. Langages d'arbres

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 $\mathcal{T}^\Sigma$.
Dans un arbre, on dit qu'un nœud est un fils gauche (resp. fils droit) si ce n'est pas la racine et que sa position termine par **0** (resp. par **1**). On considère dans ce problème les langages d'arbres suivants, 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 (autrement dit tout nœud a un fils gauche non vide si et seulement s'il a un fils droit non vide). En particulier E est complet. Le langage $L_{complet}$ est l'ensemble de tous les arbres complets.
- Un arbre est un *arbre-chaîne* si tous ses nœuds hormis la racine sont des fils gauches. En particulier E est également un arbre-chaîne. Le langage L_{chaine} est l'ensemble de tous les arbres-chaînes.
- Un arbre est *impartial* s'il a autant de nœuds qui sont fils gauches que de nœuds qui sont fils droits. En particulier E est impartial. On note $L_{impartial}$ l'ensemble de tous les arbres impartiaux.

Q. 8 Pour chacun des quatre langages L_0, $L_{complet}$, L_{chaine}, $L_{impartial}$, 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.

Q. 9 Démontrer que tout arbre complet est impartial, mais que la réciproque est fausse.

Q. 10 Démontrer que tout arbre impartial non vide a un nombre impair de nœuds.

4. Automates d'arbres descendants déterministes

Automate d'arbres descendant déterministe. Un *automate d'arbres descendant déterministe* (ou simplement *automate descendant déterministe*) sur l'alphabet Σ est un quadruplet $\mathscr{A}^{\downarrow} = (Q, q_0, F, \delta)$ 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) $F \subset Q$ est un ensemble dont les éléments sont appelés *états finaux* ;
(iv) $\delta : Q \times \Sigma \to Q \times Q$ est appelée *fonction de transition*. (On remarque que pour tout $q \in Q$ et pour tout $\alpha \in \Sigma$, $\delta(q, \alpha)$ est un couple d'états (q_g, q_d)).

Exécution acceptante. On appelle *exécution acceptante d'un automate descendant déterministe $\mathscr{A}^{\downarrow}$ sur un arbre t*, la donnée d'une fonction $\varphi : \mathcal{C}(t) \to Q$ telle que :

- $\varphi(\varepsilon) = q_0$;
- $\forall p \in \mathcal{C}_N(t)$, en notant α l'étiquette de la racine du sous-arbre $t_{|p}$ ♣, $\delta(\varphi(p), \alpha) = (\varphi(p{\cdot}\mathbf{0}), \varphi(p{\cdot}\mathbf{1}))$;
- $\forall p \in \mathcal{C}_E(t), \varphi(p) \in F$.

Autrement dit une exécution acceptante d'un automate descendant déterministe $\mathscr{A}^{\downarrow}$ sur un arbre t est une coloration de toutes les positions de l'arbre par des états de l'automate telle que :

- la racine est coloriée par l'état initial q_0 ;
- si un nœud interne étiqueté par la lettre α est colorié par un état q, alors ses fils gauche et droit sont respectivement coloriés par q_g et q_d les couleurs données par la fonction de transition δ (*i.e.* $(q_g, q_d) = \delta(q, \alpha)$) ;
- les extrémités de l'arbre sont coloriées par des états finaux.

On remarque que si une telle application φ existe, elle est nécessairement unique. On dit d'un arbre t qu'il est *reconnu* par un automate descendant déterministe $\mathscr{A}^{\downarrow}$, s'il existe une exécution acceptante de $\mathscr{A}^{\downarrow}$ sur t. L'ensemble des arbres reconnus par $\mathscr{A}^{\downarrow}$, noté $\mathcal{L}(\mathscr{A}^{\downarrow})$, est alors appelé le *langage reconnu* par $\mathscr{A}^{\downarrow}$.

Un exemple. Considérons l'automate $\mathscr{A}^{\downarrow}_{\text{ex}} = (Q, q_0, F, \delta)$ sur l'alphabet $\{\alpha_0, \alpha_1\}$ suivant :

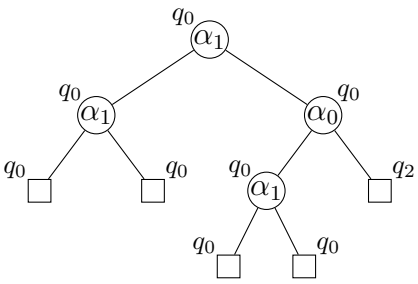
(i) $Q = \{q_0, q_1, q_2\}$
(ii) $q_0 = q_0$
(iii) $F = \{q_0, q_2\}$

♣. le sous-arbre à la position p

(iv) δ la fonction de transitions donnée dans la table ci-dessous.

	Σ	
Q	α_0	α_1
q_0	(q_0, q_2)	(q_0, q_0)
q_1	(q_1, q_1)	(q_1, q_1)
q_2	(q_1, q_1)	(q_1, q_1)

L'arbre t_0 défini page 3 admet une exécution acceptante dans l'automate $\mathscr{A}^{\downarrow}_{\text{ex}}$, celle-ci est représentée en Figure 1. La Figure 1 fournit explicitement la fonction φ, mais aussi une représentation schématique de la coloration des positions de l'arbre t_0 (y compris les extrémités) par des états de l'automate que définit φ.



$$\varphi : \begin{cases} \varepsilon & \mapsto q_0 \\ \mathbf{0} & \mapsto q_0 \\ \mathbf{00} & \mapsto q_0 \\ \mathbf{01} & \mapsto q_0 \\ \mathbf{1} & \mapsto q_0 \\ \mathbf{10} & \mapsto q_0 \\ \mathbf{100} & \mapsto q_0 \\ \mathbf{101} & \mapsto q_0 \\ \mathbf{11} & \mapsto q_2 \end{cases}$$

FIGURE 1 – Exécution acceptante φ de $\mathscr{A}^{\downarrow}_{\text{ex}}$ sur t_0.

L'automate $\mathscr{A}^{\downarrow}_{\text{ex}}$ n'admet pas d'exécution acceptante sur l'arbre $\mathsf{N}(\alpha_0, \mathsf{E}, \mathsf{N}(\alpha_1, \mathsf{E}, \mathsf{E}))$. En effet, si une telle exécution φ existait, elle vérifierait $\varphi(\varepsilon) = q_0$ et par suite $\varphi(\mathbf{1}) = q_2$ et finalement $\varphi(\mathbf{10}) = q_1$ or $q_1 \notin F$.

Q. 11 Donner un automate descendant déterministe reconnaissant le langage L_{chaine} ; aucune justification n'est demandée.

Q. 12 Montrer qu'il n'existe pas d'automate descendant déterministe qui reconnaît L_0. *Indication :* on pourra raisonner par l'absurde en supposant qu'il existe un tel automate et s'intéresser aux arbres : $\mathsf{N}(\alpha_1, \mathsf{N}(\alpha_0, \mathsf{E}, \mathsf{E}), \mathsf{N}(\alpha_1, \mathsf{E}, \mathsf{E}))$, et $\mathsf{N}(\alpha_1, \mathsf{N}(\alpha_1, \mathsf{E}, \mathsf{E}), \mathsf{N}(\alpha_0, \mathsf{E}, \mathsf{E}))$.

Implémentation des automates descendants déterministes. En OCAML, un état q_i de $Q = \{q_0, \ldots, q_{n-1}\}$ est codé par l'entier `i`. En particulier l'entier `0` représente l'état initial. L'ensemble des états finaux F est codé par un tableau de booléens `fin_dd` de taille n, tel que `fin_dd.(i)` vaut `true` si et seulement si $q_i \in F$. Enfin, les transitions sont codées par une matrice de taille $n \times m$ de couples d'entiers, telle que `trans_dd.(i).(k)` est le couple `(g, d)` vérifiant $(q_g, q_d) = \delta(q_i, \alpha_k)$.
Ainsi on représente un automate descendant déterministe en OCAML grâce au type ci-dessous.

```
type auto_dd = {
  fin_dd   : bool array ;              (* le tableau des finaux *)
  trans_dd : (int * int) array array (* la matrice des transitions *)
}
```

On donne ci-dessous une représentation en OCAML de l'automate $\mathscr{A}^{\downarrow}_{\text{ex}}$ introduit ci-avant.

```
let auto_dd_ex = {
              (* q_0 |  q_1 |  q_2  *)
  fin_dd   = [|true ; false; true |];
               (* alp_0| alp_1 *)
  trans_dd = [|[|(0, 2); (0, 0)|]; (* q_0 *)
               [|(1, 1); (1, 1)|]; (* q_1 *)
               [|(1, 1); (1, 1)|]; (* q_2*) |]}
```

Q. 13 Pour un automate descendant déterministe $\mathscr{A}^{\downarrow} = (Q, q_0, F, \delta)$ et $q \in Q$, on note $\mathscr{A}^{\downarrow}_q$ l'automate descendant déterministe (Q, q, F, δ) qui ne diffère de $\mathscr{A}^{\downarrow}$ que par son état initial. Coder une fonction `applique_desc : auto_dd -> int -> arbre -> bool` telle que, pour `a` un automate descendant déterministe $\mathscr{A}^{\downarrow}$, `q` un état $q \in Q$ et `t` un arbre, `(applique_desc a q t)` teste si $\mathscr{A}^{\downarrow}_q$ reconnaît t.

Q. 14 En utilisant `applique_desc`, coder une fonction `accepte_desc : auto_dd -> arbre -> bool` telle que `(accepte_desc a t)`, où `a` est un automate descendant déterministe $\mathscr{A}^{\downarrow}$ et `t` un arbre t, calcule si $\mathscr{A}^{\downarrow}$ reconnaît t.

5. Automates descendants et langages réguliers de mots

À tout mot $x = x_1 \dots x_l$, on associe un arbre-chaîne $\mathsf{chaîne}(x) = \mathsf{N}(x_1, \mathsf{N}(x_2, \dots \mathsf{N}(x_n, \mathsf{E}, \mathsf{E}) \dots, \mathsf{E}), \mathsf{E})$. Par convention, $\mathsf{chaîne}(\varepsilon) = \mathsf{E}$. Pour un langage de mots L, on définit le langage d'arbres $\mathsf{chaîne}(L) = \{\mathsf{chaîne}(x) \mid x \in L\}$.

Q. 15 Étant donné un arbre-chaîne $t = \mathsf{chaîne}(x)$, donner, sans justification, les ensembles $\mathcal{C}_N(t)$ et $\mathcal{C}_E(t)$. Justifier que $\mathcal{C}_N(t)$ est un langage régulier sur l'alphabet $\{\mathbf{0}, \mathbf{1}\}$.

Dans les questions suivantes (**Q. 16** à **Q. 19**), on montre que, si L est un langage régulier, $\mathsf{chaîne}(L)$ est un langage d'arbres reconnaissable par un automate d'arbre descendant déterministe. On fixe donc, pour les questions **Q. 16** à **Q. 19**, un langage L régulier.

Q. 16 Rappeler pourquoi, si L est un langage régulier, il existe un automate déterministe et complet dont le langage reconnu est L.

Soit donc un automate déterministe complet $\mathscr{A} = (\Sigma, Q, \{q_0\}, F, \delta)$ tel que $\mathcal{L}(\mathscr{A}) = L$. On assimile δ à une fonction de transition : $\delta : Q \times \Sigma \to Q$.

Q. 17 Expliquer pourquoi δ peut être assimilée à une fonction de $Q \times \Sigma \to Q$.

Soient deux états q'_1 et q'_2 qui ne sont pas dans Q. On construit l'automate d'arbre descendant déterministe $\mathscr{A}^{\downarrow} = (Q', q_0, F \cup \{q'_1\}, \delta')$ défini par :
- $Q' = Q \cup \{q'_1, q'_2\}$;
- $F' = F \cup \{q'_1\}$;
- pour tout $\alpha \in \Sigma$, pour tout $q \in Q$, $\delta'(q, \alpha) = (\delta(q, \alpha), q'_1)$, et pour tout $q \in \{q'_1, q'_2\}$ $\delta'(q, \alpha) = (q'_2, q'_2)$.

Q. 18 Montrer que $\mathcal{L}(\mathscr{A}^{\downarrow}) \subseteq \mathsf{chaîne}(L)$. Étant donné un arbre $t \in \mathcal{L}(\mathscr{A}^{\downarrow})$, on pourra considérer l'exécution acceptante φ de t dans $\mathscr{A}^{\downarrow}$, montrer qu'alors t est bien un arbre chaîne et que les lettres de la chaîne forment un mot de L.

Q. 19 Montrer que $\mathsf{chaîne}(L) \subseteq \mathcal{L}(\mathscr{A}^{\downarrow})$. Étant donné un arbre $t = \mathsf{chaîne}(w)$ pour $w \in L$, on pourra construire une exécution acceptante de t dans $\mathscr{A}^{\downarrow}$ à partir de l'exécution acceptante de w dans $\mathscr{A}$. Conclure.

Q. 20 Montrer que pour tout langage de mots L, si $\mathsf{chaîne}(L)$ est reconnu par un automate d'arbres descendant déterministe, alors L est reconnaissable. On pourra se contenter de donner un automate pour L, sans démontrer qu'il reconnaît bien L.

6. Automates d'arbres ascendants

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

(i) Q est un ensemble fini non vide dont les éléments sont appelés *états* ;
(ii) $I \subset Q$ est un ensemble dont les éléments sont appelés *état initiaux* ;
(iii) $F \subset Q$ est un ensemble dont les éléments sont appelés *états finaux* ;
(iv) $\Delta : Q \times Q \times \Sigma \to \mathscr{P}(Q)$ ♣ est appelée *fonction de transition*. (On remarque que, pour un couple d'états $(q_g, q_d) \in Q \times Q$, et une lettre $\alpha \in \Sigma$, $\Delta(q_g, q_d, \alpha)$ est un ensemble d'états).

Exécution. Exécution acceptante. On appelle *exécution d'un automate ascendant $\mathscr{A}^{\uparrow}$ sur un arbre* t, la donnée d'une fonction $\varphi : \mathcal{C}(t) \to Q$ telle que :
1) $\forall p \in \mathcal{C}_E(t), \varphi(p) \in I$;
2) $\forall p \in \mathcal{C}_N(t)$, en notant α l'étiquette de la racine du sous-arbre $t_{|p}$ ♡, $\varphi(p) \in \Delta(\varphi(p{\cdot}\mathbf{0}), \varphi(p{\cdot}\mathbf{1}), \alpha)$.
De plus, on dit que c'est une exécution *acceptante* lorsqu'elle vérifie le 3-ième point ci-dessous :
3) $\varphi(\varepsilon) \in F$.

Autrement dit une exécution d'un automate ascendant $\mathscr{A}^{\uparrow}$ sur un arbre t est une coloration de toutes les positions de l'arbre par des états de l'automate telle que :
1) les extrémités de l'arbre sont coloriées par des états initiaux ;
2) un nœud interne étiqueté par la lettre α, dont les fils gauche et droit sont respectivement coloriés par q_g et q_d est nécessairement colorié par un état autorisé par la fonction de transition, *i.e.* un état v vérifiant $q \in \Delta(q_g, q_d, \alpha)$.

De plus, on dit que c'est une exécution *acceptante*, dès lors que :
3) la racine est coloriée par un état final.

Noter que, contrairement au cas des automates descendants déterministes, quand une telle application φ existe, elle n'est pas nécessairement unique. On dit qu'un automate ascendant (Q, I, F, Δ) est *déterministe* si $|I| = 1$ et, pour tout $(q_g, q_d, \alpha) \in Q \times Q \times \Sigma$, $|\Delta(q_g, q_d, \alpha)| = 1$. Si un automate ascendant déterministe admet une exécution sur un arbre t, celle-ci est unique.

Enfin on dit d'un arbre t qu'il est *reconnu* par un automate ascendant $\mathscr{A}^{\uparrow}$, si $\mathscr{A}^{\uparrow}$ admet une exécution acceptante sur t. L'ensemble des arbres reconnus par $\mathscr{A}^{\uparrow}$, noté $\mathcal{L}(\mathscr{A}^{\uparrow})$, est alors appelé le *langage reconnu* par $\mathscr{A}^{\uparrow}$.

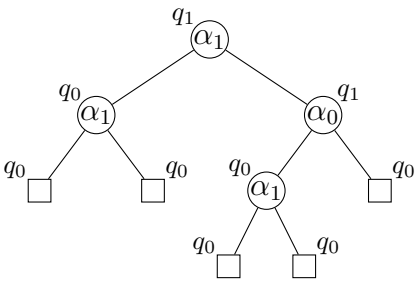
Un exemple. Considérons l'automate $\mathscr{A}_0^{\uparrow} = (Q, I, F, \Delta)$ sur l'alphabet $\{\alpha_0, \alpha_1\}$ défini par :
(i) $Q = \{q_0, q_1\}$;
(ii) $I = \{q_0\}$;
(iii) $F = \{q_1\}$;
(iv) Δ la fonction de transition donnée dans la table ci-dessous.

	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)	$\{q_1\}$	$\{q_1\}$

On observe que $\mathscr{A}_0^{\uparrow}$ ne reconnaît pas E (car $I \cap F = \{q_0\} \cap \{q_1\} = \emptyset$). L'arbre t_0 défini page 3 admet une exécution acceptante dans l'automate $\mathscr{A}_0^{\uparrow}$, celle-ci est représentée en Figure 2. La Figure 2

♣. $\mathscr{P}(X)$ désigne l'ensemble des parties de X
♡. le sous-arbre à la position p

fournit explicitement la fonction φ, mais aussi une représentation schématique du coloriage de la coloration des positions de l'arbre t_0 (y compris les extrémités) par des états de l'automate que définit φ.



$$\varphi : \begin{cases} \varepsilon & \mapsto q_1 \\ \mathbf{0} & \mapsto q_0 \\ \mathbf{00} & \mapsto q_0 \\ \mathbf{01} & \mapsto q_0 \\ \mathbf{1} & \mapsto q_1 \\ \mathbf{10} & \mapsto q_0 \\ \mathbf{100} & \mapsto q_0 \\ \mathbf{101} & \mapsto q_0 \\ \mathbf{11} & \mapsto q_0 \end{cases}$$

FIGURE 2 – Exécution acceptante φ de $\mathscr{A}_0^{\uparrow}$ sur t_0.

Notations φ_g et φ_d. Étant donné une exécution φ d'un automate (Q, I, F, Δ) sur un arbre non vide $\mathsf{N}(x, g, d)$, on pourra utiliser la notation φ_g pour désigner l'exécution dans le sous arbre gauche, à savoir $\varphi_g : \mathcal{C}(g) \to Q$ définie par : $\varphi_g(w) = \varphi(\mathbf{0}\cdot w)$. On utilisera de même φ_d. On pourra utiliser, sans devoir le démontrer que φ est une exécution de $\mathsf{N}(x, g, d)$ dans (Q, I, F, Δ) si et seulement si φ_g et φ_d sont des exécutions de (Q, I, F, Δ) sur g, resp. d, et $\varphi(\varepsilon) \in \Delta(\varphi_g(\varepsilon), \varphi_d(\varepsilon), x)$.

Q. 21 On admet que pour tout arbre t, il existe φ une exécution de $\mathscr{A}_0^{\uparrow}$ sur t. Montrer que $\mathcal{L}(\mathscr{A}_0^{\uparrow}) \subseteq L_0$. On pourra démontrer par induction que pour toute exécution φ de $\mathscr{A}_0^{\uparrow}$ sur t, $\varphi(\varepsilon) = q_1$ si et seulement s'il existe un nœud d'étiquette α_0 dans t.

Q. 22 Montrer que $L_0 \subseteq \mathcal{L}(\mathscr{A}_0^{\uparrow})$. On pourra se donner un arbre $t \in L_0$ puis considérer $\mathscr{P}$ l'ensemble non vide des positions des nœuds d'étiquette α_0 dans t et construire une exécution acceptante φ de t dans $\mathscr{A}_0^{\uparrow}$, en associant à une position p un état bien choisi selon que p est préfixe d'un élément de $\mathscr{P}$ ou non.

Q. 23 Soit L un langage d'arbres. Montrer que s'il existe un automate descendant déterministe $\mathscr{A}^{\downarrow}$ reconnaissant L, alors L est un langage d'arbres régulier. Il n'est pas attendu de justification de la correction des automates construits.

Implémentation des automates ascendants. En OCAML, 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 représenté par une liste `ini_a` sans doublons, où les états initiaux figurent dans un ordre arbitraire. L'ensemble des états finaux F est codé quant à lui par un tableau de booléens `fin_a` de taille n tel que `fin_a.(i)` vaut `true` si et seulement si $q_i \in F$. Enfin, les transitions sont codées par un tableau tridimensionnel de listes d'entiers `trans_a`, de taille $n \times n \times m$, tel que `trans_a.(i).(j).(k)` est une liste dans un ordre arbitraire des états `q` tels que $q \in \Delta(q_i, q_j, \alpha_k)$.
Ainsi on représente un automate ascendant en OCAML grâce au type ci-dessous.

```
type auto_a = {
  ini_a   : int list ;                (* les états initiaux *)
  fin_a   : bool array ;              (* le tableau des finaux *)
  trans_a : int list array array array (* les transitions *)
}
```

L'automate $\mathscr{A}_0^{\uparrow}$ peut alors être codé par la valeur suivante.

```
let aa0 = {
  ini_a    = [0];
  fin_a    = [|false ; true|];
  trans_a = [|
    [| [| [1] ; [0] |] ;
       [| [1] ; [1] |] |] ;
    [| [| [1] ; [1] |] ;
       [| [1] ; [1] |] |] }
```

Q. 24 Coder une fonction `nombre_etats_asc : auto_a -> int` prenant en argument un automate ascendant `a` et renvoyant le nombre d'états de cet automate.

Q. 25 Coder une fonction `nombre_symboles_asc : auto_a -> int` prenant en argument un automate ascendant `a` et renvoyant le nombre de symboles de l'alphabet sur lequel cet automate est défini.

Q. 26 Définir une fonction `delta : auto_a -> int list -> int list -> int -> int list` prenant en arguments un automate ascendant $\mathscr{A}^\uparrow = (Q, I, F, \Delta)$, un ensemble d'états l_g, un ensemble d'états l_d et une lettre $\alpha \in \Sigma$ et retournant l'ensemble *sans doublons*, des états q tels qu'il existe $q_g \in l_g$ et $q_d \in l_d$ tels que $q \in \Delta(q_g, q_d, \alpha)$.

Q. 27 Coder une fonction `applique_asc : auto_a -> arbre -> int list` telle que `(applique_asc a t)` où `a` est un automate ascendant $\mathscr{A}^\uparrow = (Q, I, F, \Delta)$ et `t` un arbre t, renvoie une liste sans doublon des états q pour lesquels il existe une exécution[♣] $\varphi : \mathcal{C}(t) \to Q$ avec $\varphi(\varepsilon) = q$. Si $t =$ `E`, la fonction `applique_asc` doit renvoyer la liste des états initiaux de $\mathscr{A}^\uparrow$.

Q. 28 En déduire une fonction `accepte_asc : auto_a -> arbre -> bool` telle que `(accepte_asc aa c)`, où `aa` est un automate ascendant $\mathscr{A}^\uparrow$ et `t` un arbre t, calcule si $\mathscr{A}^\uparrow$ reconnaît t.

Q. 29 Montrer qu'un langage d'arbres L est un langage d'arbres régulier si et seulement s'il existe un automate ascendant *déterministe* reconnaissant L. Il n'est pas attendu de justification de la correction des automates construits.

Q. 30 Définir une fonction OCaml `array_init : int -> (int -> 'a) -> 'a array` prenant en argument un entier `n` et une fonction `f : int -> 'a` et retournant le tableau de taille `n` suivant : `[|(f 0); (f 1); ...; (f (n-1))|]`. On pourra utiliser la fonction `Array.make : int -> 'a -> 'a array` telle que `(Array.make n x)` retourne un tableau de taille `n` initialisé avec la valeur de `x`.

Q. 31 Coder deux fonctions OCaml `id_of_partie : int list -> int` et `partie_of_id : int -> int list` réciproques l'une de l'autre, codant une bijection entre les parties de l'ensemble $[\![0, n-1]\!]$ et les entiers de 0 à $2^n - 1$, une partie étant représentée par une liste d'entiers sans doublon, dans un ordre arbitraire. On rappelle qu'en OCaml l'expression `(1 lsl i)` calcule l'entier 2^i.

Q. 32 Coder une fonction `determinise_asc : auto_a -> auto_a` prenant en argument un automate ascendant $\mathscr{A}^\uparrow$ et renvoyant un automate ascendant déterministe reconnaissant le même langage que $\mathscr{A}^\uparrow$.

Q. 33 Coder une fonction `complementaire_asc : auto_a -> auto_a` prenant en argument un automate ascendant `auto_a` reconnaissant un langage L et renvoyant un automate ascendant reconnaissant $\mathcal{T}^\Sigma \setminus L$.

Q. 34 Coder une fonction `union_asc : auto_a -> auto_a -> auto_a` prenant en arguments deux automates ascendants sur le même alphabet `aa1` et `aa2`, reconnaissant respectivement les langages L_1 et L_2 et calculant un automate ascendant reconnaissant $L_1 \cup L_2$.

♣. On ne demande pas ici à ce qu'elle soit acceptante

Q. 35 Coder une fonction `intersection_asc` prenant en arguments deux automates ascendants sur le même alphabet `aa1` et `aa2`, reconnaissant respectivement les langages L_1 et L_2 et calculant un automate ascendant reconnaissant $L_1 \cap L_2$.

Q. 36 Sans chercher à utiliser les propriétés de clôture par union, complémentation ou intersection, montrer que le langage $L_{impartial}$ n'est pas un langage d'arbres régulier.

∴ Fin du sujet ∴