TD3 - Automates finis

Exercice 1

Sur l'alphabet $\Sigma = \{a, b\}$, proposer un automate fini déterministe reconnaissant chacun des langages L suivants.

Question 1. Les mots de L contiennent au moins une fois la lettre a.

Question 2. Les mots de L contiennent au plus une fois la lettre a.

Question 3. Les mots de L contiennent un nombre pair de fois la lettre a.

Question 4. Les mots de L admettent aba pour facteur.

Question 5. Les mots de L admettent aba pour sous-mot.

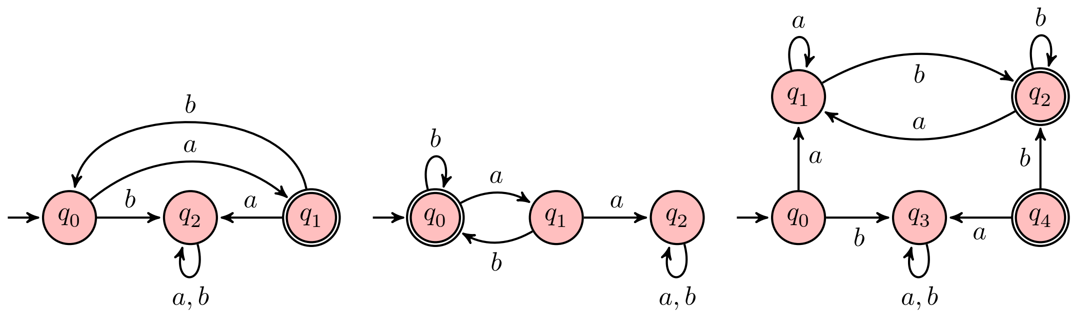
Exercice 2

L'alphabet est $\Sigma = \{a, b\}$. On considère les trois automates de la figure 1.

Question 1. Quel est le langage reconnu par l'automate représenté sur la figure 1a ?

Question 2. Émonder l'automate de la figure 1b ? Quel langage reconnait-il ?

Question 3. Identifier les états accessibles, co-accessibles et utiles de l'automate de la figure 1c. L'émonder.



(a) **(b)** **(c)**

FIGURE 1

Exercice 3

L'alphabet est $\Sigma = \{0, 1, \dots, 9, .\}$ des chiffres en base 10 et le point. Construire un automate qui reconnaît les nombres décimaux.

Exercice 4

Sur un alphabet $\Sigma = \{a, b\}$, on considère un *automate fini déterministe* $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ de fonction de transition $\delta : Q \times \Sigma \mapsto Q \cup \{\bot\}$. $\bot$ désigne un état particulier appelé état *indéfini*. En *OCaml*, l'alphabet est défini par le type `alphabet`, un mot sur cet alphabet par le type `mot` et un automate par le type `afd`. Les états d'un automate sont numérotés de 1 à n et l'état *indéfini* $\bot$ est numéroté 0.

```
type alphabet = A | B

type mot = alphabet list

type afd = {
  init    : int;
  finals  : int list;
  delta   : int * alphabet -> int;
}
```

```
let a1 = {init = 1; finals = [3];
  delta = function
    | (1, a) -> 2
    | (1, b) -> 1
    | (2, a) -> 1
    | (2, b) -> 3
    | (3, a) -> 1
    | _ -> 0
}
```

Question 1. Dessiner l'automate fini déterministe $\mathcal{A}_1$ défini par `a1`.

Question 2. Écrire une fonction `delta_star : int -> mot -> afd -> int` associée à la fonction de transition δ^* des mots.

Question 3. Écrire une fonction `reconnait : mot -> afd -> bool` qui teste l'appartenance d'un mot à un langage reconnu par un automate. Estimer sa complexité.

Exercice 5

On considère un *automate fini non-déterministe* $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ sur un alphabet Σ. On désigne par n son nombre d'états et par p le nombre de ses transitions. I est l'ensemble, non nécessairement réduit à un seul élément, des états initiaux. On rappelle qu'un état $q \in Q$ de l'automate est :

- *accessible* s'il existe un état initial $q_i \in I$ et un mot $w \in \Sigma^*$ tel que $q_i \xrightarrow{w}{}^*_{\mathcal{A}} q$.
- *co-accessible* s'il existe un état final $q_f \in F$ et un mot $w \in \Sigma^*$ tel que $q \xrightarrow{w}{}^*_{\mathcal{A}} q_f$.

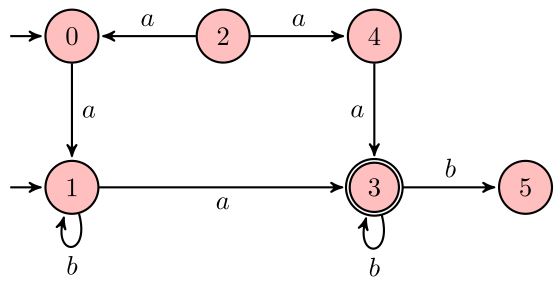
Un mot w est *reconnu* par l'automate $\mathcal{A}$ s'il existe un état initial $q_i \in I$ et un état acceptant $q_f \in F$ tels que $q_i \xrightarrow{w} q_f$. Un état q qui participe à la reconnaissance d'un mot vérifie $q_i \xrightarrow{w'}{}^*_{\mathcal{A}} q \xrightarrow{w''} q_f$ est à la fois accessible et co-accessible. En supprimant tous les états non accessibles et non co-accessibles de l'automate ainsi que des transitions partant ou arrivant de ces états inutiles, on obtient l'*automate émondé* $\mathcal{A}_e$ qui reconnaît le même langage que l'automate $\mathcal{A}$: $\mathcal{L}(\mathcal{A}_e) = \mathcal{L}(\mathcal{A})$.

Ce sujet étudie deux algorithmes de parcours en largeur d'un graphe qui calculent l'émondé d'un automate. Un automate fini non-déterministe est représenté par le type suivant.

```
type af = {
  alphabet  : char array;                (* tableau alphabet *)
  etats     : int;                       (* nombre d'états *)
  init      : int list;                  (* états initiaux *)
  finals    : int list;                  (* états acceptants *)
  trans     : (int * char * int) list;   (* transitions *)
}
```

Les états de l'automate sont représentés par des entiers de $[\![0, n-1]\!]$ où `etats` est le nombre d'états n de l'automate, `init` est la liste *triée* des états initiaux, `finals` est la liste *triée* des états acceptants, `trans` est la liste des transitions de l'automate. L'automate `a1` ci-dessous servira de test pour les fonctions.

```
let a1 = {
  alphabet  = [|'a'; 'b' |];
  etats     = 6;
  init      = [0; 1];
  finals    = [3];
  trans     = [(0, 'a', 1); (1, 'b', 1); (1, 'a', 3); (2, 'a', 0);
               (2, 'a', 4); (3, 'a', 3); (4, 'b', 3); (3, 'b', 5);]
}
```



Liste de successeurs

Cette partie travaille avec les listes des successeurs des états pour déterminer les états accessibles, co-accessibles puis calculer l'automate émondé. Les listes d'états représentent des *ensembles*. Un état n'y apparaît au plus qu'une fois et les listes d'états sont toujours *triées* par ordre croissant.

Question 1.
□ **1.1.** Écrire une fonction `add : 'a -> 'a list -> 'a list` qui ajoute un état à une liste d'états.
□ **1.2.** Écrire une fonction `merge : 'a list -> 'a list -> 'a list` qui réalise la fusion ordonnées de deux listes.
□ **1.3.** Écrire une fonction `eql : 'a list -> 'a list -> bool` qui teste si deux listes triées sont égales.
□ **1.4.** Écrire une fonction `intersect : 'a list -> 'a list -> 'a list` qui calcule l'intersection de deux listes.
□ **1.5.** Estimer leurs complexités.

Question 2. On considère un état `q` et un automate `af`. Écrire deux fonctions `succ : int -> af -> int list` et `pred : int -> af -> int list` telles que `(succ q af)` et `(pred q af)` renvoient les listes triées des successeurs et des prédécesseurs de `q` dans `af`. Estimer leurs complexités.

Question 3. La fonction `gamma` désigne soit la fonction `succ`, soit la fonction `pred`. Écrire une fonction `etend : int list -> af -> (int -> af -> int list)-> int list` telle que `(extend lst af gamma)` ajoute à la liste d'états `lst`, les états $\{\texttt{q}\} \cup \texttt{gamma}(\texttt{q}, \texttt{af})$ où $\texttt{q} \in \texttt{lst}$. Justifier sa terminaison.

Question 4. En déduire une fonction `access : af -> (int -> int -> int list)-> int list -> int list` telle que `(access af gamma lst)` renvoie la liste des états accessibles à partir de la liste d'états `lst` en itérant la fonction `gamma`.

Question 5. Écrire une fonction `rm_trans : 'a list -> ('a * 'b * 'a)list -> ('a * 'b * 'a)list` telle que `(supprime_trans etats transitions)` renvoie la liste des transitions dans laquelle on a supprimé toutes les transitions faisant intervenir un état qui n'est pas dans la liste `etats`.

Question 6. En déduire une fonction `trim : af -> af` qui calcule l'automate émondé.

Algorithme de Warshall

L'automate est représenté par un graphe orienté dont les sommets sont les états numérotés de 0 à $(n-1)$ et les arêtes sont les transitions. Pour un tel graphe orienté, on définit la *matrice d'adjacence* $C = (C[i,j])_{0\leqslant i,j\leqslant n-1}$ de booléens en posant $C[i,j]$ égal à `true` lorsque $i = j$ ou lorsqu'il existe une transition de l'état i vers l'état j, égal à `false` sinon.

Question 7. Écrire une fonction `mat_adj : af -> bool array array` qui calcule la matrice d'adjacence d'un automate.

L'algorithme de Warshall consiste à calculer une suite de matrices $C_0, C_1 \dots, C_{n-1}$ telle que $C_k[i,j] = \texttt{true}$ si et seulement s'il existe un chemin partant de l'état i, arrivant à l'état j en ne passant que par des états intermédiaires dans $[\![0,k]\!]$. On initialise l'algorithme avec la matrice d'adjacence C du graphe.

Question 8. En supposant construite la matrice C_{k-1}, montrer que $C_k[i,j] = \texttt{true}$ si et seulement si l'une des deux conditions suivantes est vérifiée :

- $C_{k-1}[i,j] = \texttt{true}$
- $C_{k-1}[i,k] = \texttt{true}$ et $C_{k-1}[k,j] = \texttt{true}$.

Question 9. En supposant construite la matrice C_{k-1}, montrer que :
□ **9.1.** $C_k[i,k] = \texttt{true}$ si et seulement si $C_{k-1}[i,k] = \texttt{true}$;
□ **9.2.** $C_k[k,j] = \texttt{true}$ si et seulement si $C_{k-1}[k,j] = \texttt{true}$.

Question 10.
□ **10.1.** Écrire une fonction `matrice_accessibilite : af -> bool array array` qui renvoie la matrice D avec $D[i,j] = \texttt{true}$ si et seulement s'il existe un chemin de l'état i à l'état j.
□ **10.2.** Estimer la complexité du calcul de D en supposant connue la matrice d'adjacence et ne tenant compte que des opérations booléennes.

Question 11.
□ **11.1.** Écrire une fonction `deletat : int -> bool array array -> int list` telle que si `c` est la matrice d'accessibilité du graphe, `(deletat i c)` renvoie la liste triée des états accessibles depuis l'état `i`. Une fonction récursive auxiliaire peut être utile.
□ **11.2.** Écrire une fonction `accessibles : int list -> bool array array -> int list` qui renvoie la liste des états accessibles à partir des états d'une liste `lst`.
□ **11.3.** En déduire la fonction `etats_acc : af -> int list` qui retourne la liste des états accessibles de l'automate.
On programme la recherche des états co-accessibles et l'émondage de l'automate comme en première partie.