TP 3 : Automates finis

Rappel Compilation OCaml :

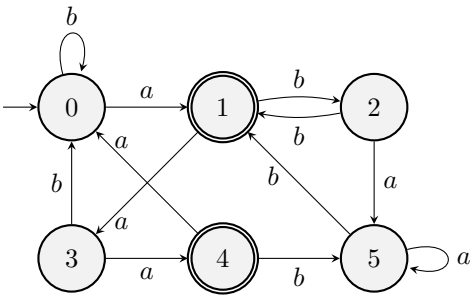
Compilation d'un fichier `main.ml` en un exécutable `nom_executable`.

```
> ocamlc -o nom_executable main.ml
```

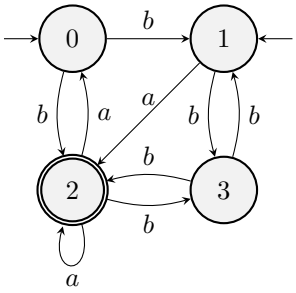
Mise en place

1. Veuillez trouver dans le répertoire partagé le fichier `automates.ml`. Examinez son contenu.

   Observez en particulier que dans le type `automate`, l'ensemble des transitions est représenté par une table de hachage : à chaque paire `(q,b)`, elle associe la liste des états dans lesquels on peut arriver en lisant la lettre `b` dans l'état `q` de l'automate.
2. Écrivez une fonction `est_deterministe : 'a automate -> bool` qui renvoie `true` si et seulement si l'automate est déterministe.
3. On fournit le code de l'automate `a0`, représenté ci-dessous :



   Vérifiez qu'il est bien déterministe à l'aide de la fonction que vous venez d'écrire.
4. En vous inspirant du code écrit pour l'automate `a0`, représentez l'automate non-déterministe `a1` donné ci-dessous, dans votre code :



5. Vérifiez à l'aide de votre fonction que cet automate est bien non-déterministe.

Parties de $[\![0, N-1]\!]$

Pour lire un mot dans un automate non-déterministe, on a besoin de pouvoir considérer des ensembles d'états, puisque lire un même mot peut nous amener dans plusieurs états distincts.

Dans cette section, on considère l'intervalle d'entiers $[\![0, N-1]\!]$. Une partie P de cet intervalle est représentée par un tableau t de N booléens tel que $t[i]$ est vrai si et seulement si $i \in P$.

6. Écrivez une fonction `appartient : int -> bool array -> bool` telle que `appartient i p` teste si l'entier `i` est dans la partie représentée par le tableau booléen `p`.
7. Écrivez une fonction `vide : int -> bool array` telle que `vide n` construit la partie vide de l'intervalle $[0, n-1]$
8. Écrivez une fonction `est_vide : bool array -> bool` testant si une partie représentée par un tableau de booléens est vide.
9. Écrivez une fonction `intersection : bool array -> bool array -> bool array` réalisant l'intersection de deux parties de $[\![0, N-1]\!]$ dans un nouveau tableau.
10. Écrivez une fonction `ajoute : bool array -> bool array -> unit` telle que `ajoute e1 e2` qui ajoute dans la partie représentée par `e1` les éléments de la partie représentée par `e2`. (À la fin, `e1` est l'union des deux parties)
11. Écrivez une fonction `construit : int list -> int -> bool array` telle que `construit l n` renvoie un tableau de taille `n` représentant l'ensemble des éléments contenus dans la liste `l`.

Lecture d'un mot

On va maintenant utiliser la section précédente. En effet, on sait que lire un mot dans un automate non-déterministe peut nous amener dans plusieurs états. Si on appelle Q l'ensemble des états d'un automate, on va considérer qu'on lit des mots en gardant trace des états où l'on peut être grâce à un tableau de booléens qui représente une partie de Q.

12. Écrivez une fonction `lire_lettre_etat : 'a automate -> int -> 'a -> bool array` telle que l'appel `lire_lettre_etat a q b` renvoie la partie correspondant à l'ensemble des états où l'on peut se trouver après avoir lu `b` dans l'état `q` de l'automate `a`.
13. Écrivez une fonction `lire_lettre_partie : 'a automate -> bool array -> 'a -> bool array` telle que `lire_lettre_partie a p b` renvoie la partie correspondant à l'ensemble des états obtenu en lisant `b` dans chaque état de l'ensemble `p`.
14. Écrivez une fonction `suivant_partie : 'a automate -> bool array -> 'a -> bool array` telle que `suivant a p b` renvoie l'ensemble des états où l'on peut se trouver après avoir lu `b` depuis la partie `p` des états de l'automate `a`.
15. Écrivez une fonction `lire_mot :'a automate -> bool array -> 'a mot -> bool array` telle que pour un automate `a`, l'appel `lire_mot a p u` effectue la lecture du mot `u` depuis tous les états de la partie `p`.
16. Écrivez une fonction `accepte_mot : 'a automate -> 'a mot -> bool` qui teste si un mot est accepté par un automate non-déterministe.

Déterminisation

Pour ne pas répéter les opérations sur les parties de la section précédente à chaque test, on va tenter de déterminiser nos automates.

On rappelle que si $\mathcal{A} = (Q, \Sigma, I, F, \delta)$, alors l'automate déterminisé a pour ensemble d'états l'ensemble des parties de Q. Chacun de ces états, c'est-à-dire chaque partie P de Q, peut être représenté par un tableau de booléens d'après ce qui précède. On peut lire ce tableau de booléen comme un nombre en binaire.

17. Écrire une fonction `vers_entier : bool array -> int` qui prend un tableau de booléens, le lit comme si c'était un nombre en binaire (avec les bits de poids fort en premier), et renvoie l'entier correspondant. Par exemple, le tableau `[|true;true;false;false|]` peut être interprété comme le nombre binaire 1100, et la fonction renvoie alors 12.
18. Écrire une fonction `vers_partie : int -> int -> bool array`, telle que `vers_partie p n` encode en binaire l'entier `p`, au moyen d'un tableau de booléens de taille `n`.
19. Écrire une fonction `puissance : int -> int -> int`, telle que `puissance p n` calcule p^n.

20. (*) Écrire une fonction `determinise : 'a automate -> 'a automate`, qui construit un automate déterministe à partir de l'automate fourni en argument.

    Indice 1 : Gardez traces des états de l'automate déterminisé déjà traités à l'aide d'un tableau de booléens de la bonne taille. Commencez par créer dans cette fonction les attributs de l'automate déterminisé les plus faciles à trouver (le nombre d'états et l'ensemble des états initiaux)/ Créez dans cette fonction une table de hachage qui sera utilisée pour stocker les transitions de l'automate déterminisé. Elle sera initialement vide.

    Indice 2 : Utilisez une fonction auxiliaire récursive qui prend en argument :

    - l'état `q` qu'on est en train de traiter lors de l'appel.
    - l'indice `j` correspondant au numéro de la lettre de l'alphabet qu'on s'apprête à considérer.
    - la liste `reste` des états de l'automate déterminisé qui restent encore à traiter. Elle est utilisée comme accumulateur dans la fonction auxiliaire.
    - la liste `accept` des états acceptants de l'automate déterminisé que nous avons déjà rencontrés. Elle est aussi utilisée comme accumulateur dans la fonction auxiliaire.

    Déterminez convenablement l'appel récursif à faire en fonction des arguments.

Annexes

Tables de hachage

Pour manipuler des tables d'association (dictionnaires):

- `Hashtbl.create n` Renvoie une table de hachage de taille n.
- `Hashtbl.add table cle valeur` Ajoute l'association clé $\rightarrow$ valeur à la table de hachage.
- `Hashtbl.replace table cle valeur` Remplace la valeur courante associée à `cle` dans la table par `valeur`.
- `Hashtbl.mem table cle` renvoie `true` ssi la cle a une valeur associée dans la table.
- `Hashtbl.find table cle` renvoie la valeur associée à la clé dans la table.
- `Hashtbl.find_opt table cle` renvoie une option sur la valeur associée à la clé dans la table (c'est-à-dire `None` s'il n'en existe pas, et `Some v` si la valeur `v` a été trouvée.

Affichage

Graphviz est un logiciel de visualisation de graphe que nous serons amenés à réutiliser lors du prochain chapitre sur les algorithmes des graphes. Le logiciel prend en entrée une description textuel de graphe sous la forme d'un fichier `graphe.dot` et crée une image de ce graphe (sous format svg, png ou pdf). Graphviz est installable (voir déjà installé) sur les machines Linux mais a aussi une interface en ligne: `https://graphviz.christine.website`.

Pour représenter un automate, on utilisera un graphe orienté: un `digraph`. Dans ce graphe, le format `.dot` attend une liste d'arcs `s1 -> s2;`. Il déduit lui-même la liste des sommets depuis son ensemble d'arcs mais on peut aussi lui préciser l'existence d'un sommet avec une simple déclaration `s1;`.

Pour étiqueter un arc, par exemple avec la lettre a, on précède la déclaration de l'arc d'un `edge [label = "a"];`. Pour changer le style d'un sommet, on précède la déclaration de ce sommet d'un `node [shape = "circle"]` (sommet avec un simple cercle) ou `node [shape = "doublecircle"]` (sommet avec un double cercle, pour les états acceptants).

Les états d'entrée seront créé en créant un sommet "fantôme" sans contour (`shape = "none"`) et sans étiquette.

La description de l'automate $\mathcal{A}_0$ est par exemple:

```
digraph G {
node [shape = "doublecircle"];
1;
4;
node [shape = "circle"];

edge [label = "a"];
0 -> 1;
4 -> 0;
1 -> 3;
3 -> 4;
2 -> 5;
5 -> 5;

edge [label = "b"];
0 -> 0;
3 -> 0;
1 -> 2;
2 -> 1;
4 -> 5;
5 -> 1;

edge [label = ""];
node [shape = "none", label = ""];
in0 -> 0;
}
```

Graphviz construira alors le graphe suivant avec son moteur dot.