TP 3 : Langages réguliers, expressions régulières

Environnement

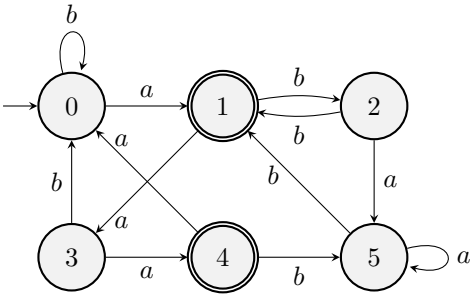
Nous travaillerons en mode interprété pour ce TP. Nous vous laissons le choix de l'environnement, selon vos habitudes. Vous pouvez utiliser :

- L'interpréteur en ligne BetterOCaml : https://BetterOCaml.ml/
- Emacs, muni du mode Tuareg.

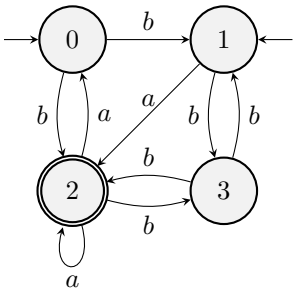
Mise en place

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. Ecrivez une fonction `est_deterministe : 'a automate -> bool`, qui vérifie si un automate est déterministe ou non.
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'état, 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. Ecrivez une fonction `appartient : int -> bool array -> bool` tel que `appartient i p` teste si l'entier `i` est dans la partie représentée par le tableau booléen `p`.
7. Ecrivez une fonction `vide : int -> bool array` tel que `vide n` construit la partie vide de l'intervalle $[0, n-1]$
8. Ecrivez une fonction `est_vide : bool array -> bool` testant si une partie représentée par un tableau de booléens est vide.
9. Ecrivez une fonction `union : bool array -> bool array -> bool array` (resp. `intersection : bool array -> bool array -> bool array`) réalisant l'union (resp. l'intersection) de deux parties de $[\![0, N-1]\!]$ dans un nouveau tableau.
10. Ecrivez 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`. (A la fin, `e1` est l'union des deux parties)
11. Ecrivez une fonction `construit : int list -> int -> bool array` tel 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. Ecrivez une fonction `lire_lettre_etat : 'a automate -> int -> 'a -> bool array` tel que `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. Ecrivez une fonction `lire_lettre_partie : 'a automate -> bool array -> 'a -> bool array` tel 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. Ecrivez une fonction `suivant_partie : 'a afnd -> bool array -> letter -> bool array` tel 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. Ecrivez une fonction `lire_mot : 'a automate -> bool array -> 'a mot -> bool array` telle que `lire_mot a p u` effectue la lecture d'un mot depuis tous les états de la partie `p`.
16. Ecrivez 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. Ecrire une fonction `vers_entier : bool array -> int` qui prend un tableau de booléens, l'analyse comme si c'était un nombre en binaire, et renvoie l'entier correspondant. Par exemple, le tableau `[|true;true;false;fa` peut être interprété comme le nombre en binaire 1100, et la fonction précédente renvoie alors 12.
18. Ecrire 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. Ecrire une fonction `puissance : int -> int -> int`, telle que `puissance p n` calcule p^n.
20. (*) Ecrire 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.