TP2 - Automates en OCaml

À retenir :

- modélisation d'un automate sans ε-transition en `OCaml`
- test d'acceptation d'un mot par un automate, même non déterministe
- déterminisation d'automates
- implémentation du type ensemble par des listes

Dans tout ce TP, les automates ont pour ensemble d'états un intervalle d'entiers de la forme $[\![0, n-1]\!]$ et pour alphabet les m premières lettres de l'alphabet latin pour un certain $m \in \mathbb{N}$, de sorte qu'on l'identifie à $[\![0, m-1]\!]$. Par exemple, si $m = 3$, l'alphabet est $\{a, b, c\}$.

1. Définir une fonction `int_of_letter : char -> int` prenant en argument un caractère `c` de l'alphabet latin et retournant son numéro.
Par exemple, (`int_of_letter 'a'`) vaut 0 et (`int_of_letter 'e'`) vaut 4. Si le caractère pris en argument n'est pas une lettre minuscule de l'alphabet latin, cette fonction lèvera une exception.
Indication. La fonction `int_of_char : char -> int` retourne le code ASCII d'un caractère.

I. Automates déterministes complets

Pour représenter un automate déterministe complet, on définit une structure contenant 3 champs :

- un champ `init` qui est un entier de $[\![0, n-1]\!]$ représentant l'état initial
- Un champ `fin` qui est une liste d'entiers de $[\![0, n-1]\!]$ représentant les états finaux
- Un champ `trans` qui est une matrice de transition de taille $n \times m$. La case `trans.(i).(j)` représente l'état que l'on atteint depuis l'état i en lisant la j-ème lettre de l'alphabet.

```
type automate_d = {
    init : int; (* état initial *)
    fin : int list; (* la liste des états finaux *)
    trans : int array array; (* la table des transitions *)
}
```

2. Représenter graphiquement l'automate déterministe complet représenté par la valeur `OCaml` ci contre.

```
let auto_1 = {
    init = 0;
    trans =
    (* | a | b | c | d | e |*)
    [|[| 1 ; 0 ; 0 ; 4 ; 4 |]; (* état 0 *)
      [| 2 ; 4 ; 4 ; 1 ; 4 |]; (* état 1 *)
      [| 3 ; 2 ; 2 ; 4 ; 4 |]; (* état 2 *)
      [| 0 ; 4 ; 4 ; 4 ; 3 |]; (* état 3 *)
      [| 4 ; 4 ; 4 ; 4 ; 4 |]; (* état 4 *)
    |];
    fin = [2]
}
```

3. Définir une fonction `delta : automate_d -> int -> char -> int` prenant en arguments : un automate, un état q, une lettre l et retournant $\delta(q, l)$.

On définit la fonction $\delta^* : Q \times \Sigma^* \to Q$ comme suit :

$$\delta^*(q, w) = \begin{cases} q & \text{si } w = \varepsilon, \\ \delta(\delta^*(q, u), x) & \text{si } w = ux \quad (u \in \Sigma^*,\ x \in \Sigma). \end{cases}$$

Ainsi, $\delta^*(q, w)$ correspond à l'état obtenu depuis q en lisant w.

4. Définir une fonction `delta_etoile : automate_d -> int -> string -> int` prenant en arguments un automate, un état q, un mot w et retournant $\delta^*(q, w)$.

5. En déduire une fonction `accepte : automate_d -> string -> bool` prenant en arguments un automate a, un mot w et retournant si le mot w est accepté par l'automate a.
Indication. On pourra utiliser `List.mem : 'a -> 'a list -> bool`

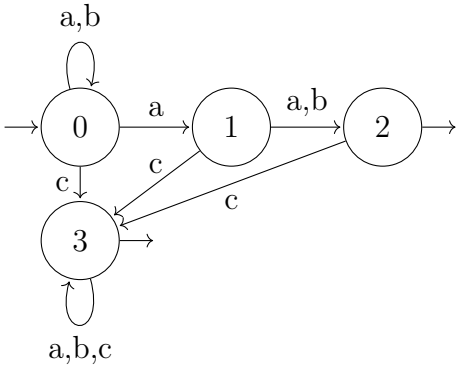
II. Automates non déterministes sans ε-transitions

On s'intéresse maintenant à la manipulation d'automates non nécessairement déterministes et ni complets. De tels automates seront représentés au moyen d'une structure contenant 3 champs :

- Un champ `init` qui est une liste d'entiers de $[\![0, n-1]\!]$, sans doublons, représentant les états initiaux.
- Un champ `init` qui est une liste d'entiers de $[\![0, n-1]\!]$, sans doublons, représentant les états finaux.
- Un champ `trans` qui est une matrice de dimensions $n \times m$. La case `trans.(i).(j)` de la matrice représente l'ensemble des états que l'on atteint depuis l'état i en lisant la j-ème lettre de l'alphabet par une liste sans doublons (possiblement vide).

```
type automate_d = {
    init : int list; (* la liste des états initiaux *)
    fin : int list; (* la liste des états finaux *)
    trans : int list array array; (* la table des transitions *)
}
```

6. Donner la représentation `OCaml` de l'automate suivant :



7. Définir une fonction `est_deterministe: automate_nd -> bool` prenant en argument un automate et testant si celui-ci est déterministe.

8. Définir une fonction `enleve_doublons : 'a list -> 'a list` prenant en argument une liste et en retirant les doublons. On acceptera une complexité quadratique en la taille de la liste.

9. Définir une fonction `delta_singleton : automate_nd -> int -> char -> int list` prenant en arguments un automate a, un état q, une lettre l et retournant l'ensemble $\delta(q, l)$ sous la forme d'une liste sans doublons.

10. En déduire une fonction `delta : automate_nd -> int list -> char -> int list` prenant en arguments un automate a, un ensemble d'états s, une lettre l et retournant l'ensemble des états accessibles depuis un état de s en lisant la lettre l, sous la forme d'une liste sans doublons.

11. Définir une fonction `delta_etoile : automate_nd -> int list -> string -> int list` prenant en arguments un automate a, un ensemble d'états s, un mot w et retournant l'ensemble des états accessibles depuis un état de s en lisant le mot w, sous la forme d'une liste sans doublons.

12. En déduire une fonction `accepte : automate_nd -> string -> bool` prenant en arguments un automate, un mot et retournant si le mot w est accepté par l'automate.

III. Déterminisation

13. Proposer une fonction `determinise : automate_nd -> automate_d` appliquant l'algorithme de déterminisation à l'automate qui lui est passé en argument.
Indications. On rappelle que les sous-ensembles de $[\![0, n-1]\!]$ peuvent être représentés par n entiers de $\{0, 1\}$, et qu'une telle séquence peut aussi représenter un entier de l'intervalle $[\![0, 2^n - 1]\!]$ grâce à l'écriture en binaire.
Ainsi les sous-ensembles de $[\![0, n-1]\!]$ et les entiers de $[\![0, 2^n - 1]\!]$ sont en bijection. On pourra définir deux fonctions `int_of_intlist : int list -> int` et `intlist_of_int : int -> int list` calculant cette bijection et sa réciproque.
Par exemple 22 s'écrit 10110 en binaire. Ainsi `int_of_intlist [1; 2; 4]` vaut `22` et `intlist_of_int 22` vaut `[1; 2; 4]`.