

Ce TP est à effectuer en OCaml, sous Visual Studio Code, dans Codespaces ou sur un ordinateur personnel. Compléter le fichier `tp2.ml` fourni. Pour exécuter les définitions dans `utop`, sélectionner les lignes puis appuyer sur Maj + Entrée. Ajouter `;;` pour terminer une saisie dans `utop`. Interrompre une boucle infinie avec Ctrl + C.

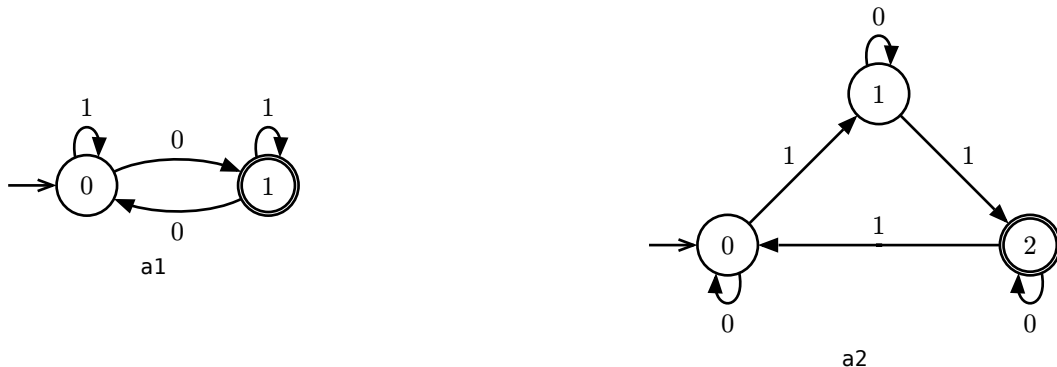
Automate déterministe

On représente un ADC par le type suivant :

```
type adc = {
  initial : int;
  finaux : int list;
  delta : int array array
}
```

Les états sont numérotés de 0 à $n - 1$ et les lettres sont des entiers : a est représenté par 0, b par 1, etc. Pour un automate a , $a.\text{delta}.(i).(j)$ est l'état atteint depuis l'état i en lisant la lettre j . Le nombre d'états est `Array.length a.delta`.

1. Définir en OCaml les automates `a1` et `a2` suivants :



```
let a1 = {
  initial = 0;
  finaux = [1];
  delta = [| [| 1; 0 |]; [| 0; 1 |] |]
}

let a2 = {
  initial = 0;
  finaux = [2];
  delta = [| [| 0; 1 |]; [| 1; 2 |]; [| 2; 0 |] |]
}
```

2. Définir `delta_etoile` : `adc -> int -> int list -> int` telle que `delta_etoile a q m` renvoie l'état atteint depuis q après lecture de m , c'est-à-dire $\delta^*(q, m)$. Vérifier avec `a1`.

```
let rec delta_etoile a q u = match u with
| [] -> q
| x :: suite -> delta_etoile a a.delta.(q).(x) suite
```

3. Définir `accepte` : `adc -> int list -> bool` telle que `accepte a m` détermine si m est reconnu par a . Vérifier avec `a1`. Quelle est sa complexité ?

```
let accepte a u =
  List.mem (delta_etoile a a.initial u) a.finaux
```

La lecture du mot coûte $O(|m|)$ et la recherche de l'état final coûte $O(|F|)$. La complexité totale est $O(|m| + |F|)$.

4. Définir `complementaire` : `adc -> adc` qui construit un automate reconnaissant le complémentaire du langage reconnu par a . Vérifier avec `a1`.

```
let complementaire a =
  let rec finaux q =
    if q < 0 then []
    else if List.mem q a.finaux then finaux (q - 1)
```

```
    else q :: finaux (q - 1) in
  { initial = a.initial; finaux = finaux (Array.length a.delta - 1); delta = a.delta }
```

5. Définir `accessibles : adc -> int list` qui détermine les états accessibles de `a` par un parcours en profondeur. Vérifier sur des exemples.

Indication. Utiliser un tableau `vus`, puis visiter récursivement chaque successeur non encore visité depuis l'état initial.

```
let accessibles a =
  let vus = Array.make (Array.length a.delta) false in
  let rec visiter q =
    vus.(q) <- true;
    for x = 0 to Array.length a.delta.(q) - 1 do
      let r = a.delta.(q).(x) in
      if not vus.(r) then visiter r
    done in
  visiter a.initial;
  let rec liste q =
    if q < 0 then []
    else if vus.(q) then q :: liste (q - 1)
    else liste (q - 1) in
  liste (Array.length a.delta - 1)
```

OCaml

6. En déduire `vide : adc -> bool` qui détermine si le langage reconnu par `a` est vide.

```
let vide a =
  not (List.exists (fun q -> List.mem q a.finaux) (accessibles a))
```

OCaml

Automate produit

7. Écrire `inter : adc -> adc -> adc` pour construire un automate reconnaissant l'intersection des langages reconnus par `a` et `b`, sur le même alphabet. Construire l'automate produit à np états, où n et p sont les nombres d'états de `a` et `b`. Numéroté l'état (i, j) par $i \times p + j$. Vérifier avec `a1` et `a2`.

```
let inter a b =
  let n = Array.length a.delta in
  let p = Array.length b.delta in
  let s = Array.length a.delta.(0) in
  let d = Array.make_matrix (n * p) s 0 in
  let f = ref [] in
  for i = 0 to n - 1 do
    for j = 0 to p - 1 do
      for x = 0 to s - 1 do
        d.(i * p + j).(x) <- a.delta.(i).(x) * p + b.delta.(j).(x)
      done;
      if List.mem i a.finaux && List.mem j b.finaux then
        f := (i * p + j) :: !f
    done
  done;
  { initial = a.initial * p + b.initial; finaux = !f; delta = d }
```

OCaml

8. En déduire `inclus : adc -> adc -> bool` qui teste l'inclusion des langages reconnus par `a` et `b`.

On utilise $\mathcal{L}(A) \subset \mathcal{L}(B) \iff \mathcal{L}(A) \cap \overline{\mathcal{L}(B)} = \emptyset$.

```
let inclus a b =
  vide (inter a (complementaire b))
```

OCaml

9. En déduire `equivalent : adc -> adc -> bool` qui teste l'égalité des langages reconnus par `a` et `b`.

```
let equivalent a b =
  inclus a b && inclus b a
```

OCaml