Table of Contents

TP 4 - Programmation pour la préparation à l'agrégation maths option info

TP 4 : Automates et langages réguliers.

- En OCaml.

```
In [32]: let print = Printf.printf;;
         Sys.command "ocaml -version";;

Out[32]: val print : ('a, out_channel, unit) format → 'a = <fun>

         The OCaml toplevel, version 4.04.2

Out[32]: - : int = 0
```

Remise en forme : listes associatives

Certaines de ces fonctions sont dans la bibliothèque standard dans le module `List`, avec des fonctions contenant `assoc` dans leur nom :

In [33]:
```
List.mem;;  (* appartient *)
```
Out[33]:
```
- : 'a → 'a list → bool = <fun>
```

In [34]:
```
List.assoc;;  (* trouve *)
```
Out[34]:
```
- : 'a → ('a * 'b) list → 'b = <fun>
```

In [35]:
```
List.mem_assoc;;  (* existe *)
```
Out[35]:
```
- : 'a → ('a * 'b) list → bool = <fun>
```

In [36]:
```
List.remove_assoc;;  (* supprime *)
```
Out[36]:
```
- : 'a → ('a * 'b) list → ('a * 'b) list = <fun>
```

Exercice 1 : `appartient`

On propose plusieurs implémentations, toutes similaires mais de complexités différentes. Je vous laisse trouver les différences de comportement (lesquelles sont tout le temps linéaire, au mieux $\mathcal{O}(1)$ etc).

In [37]:
```
(* En O(n) pour une liste de taille n (pire cas), en O(1) meilleur cas. *)
let rec appartient (x:'a) (l:'a list) : bool =
    match l with
    | [] -> false
    | y :: _ when x = y -> true
    | _ :: q -> appartient x q
;;
```
Out[37]:
```
val appartient : 'a → 'a list → bool = <fun>
```

In [38]:
```
let liste1 = [ 1; 2; 3 ];;
let couple1 = (1, 2, 3) ;;
```
Out[38]:
```
val liste1 : int list = [1; 2; 3]
```
Out[38]:
```
val couple1 : int * int * int = (1, 2, 3)
```

In [39]:
```
(* En O(n) pour une liste de taille n (pire cas), en O(1) meilleur cas. *)
let rec appartient (x:'a) (l:'a list) : bool =
    match l with
    | [] -> false
    | y :: q -> (x = y) || appartient x q
;;
```
Out[39]:
```
val appartient : 'a → 'a list → bool = <fun>
```

In [40]:
```
(* En O(n) pour une liste de taille n (pire cas), en O(n) meilleur cas. *)
let rec appartient (x:'a) (l:'a list) : bool =
   match l with
   | [] -> false
   | y :: q -> appartient x q || x = y
;;
```
Out[40]:
```
val appartient : 'a → 'a list → bool = <fun>
```

In [41]:
```
let appartient = List.mem;;
```
Out[41]:
```
val appartient : 'a → 'a list → bool = <fun>
```

In [42]:
```
assert (appartient 3 [1;2;3;4;5]) ;;
assert (not (appartient 9 [1;2;3;4;5])) ;;
```

Out[42]:
```
- : unit = ()
```

Out[42]:
```
- : unit = ()
```

Exercice 2 : `insere`

On a envie d'écrire rapidement cela :

In [43]:
```
let insere (k:'a) (v:'b) (l: ('a*'b) list) : ('a*'b) list =
    (k,v) :: l
;;
```

Out[43]:
```
val insere : 'a → 'b → ('a * 'b) list → ('a * 'b) list = <fun>
```

Mais on peut réfléchir à la sémantique que l'on shouaite donner à cette fonction `insere` : si la clé `k` est déjà présente, doit-on échouer, ou ajouter une deuxième valeur associée à la même clé, ou écraser la valeur déjà associée à `k` ? Vous pouvez essayer d'implémenter chacun des variantes !

On construit un exemple de petite liste associative :

In [44]:
```
let justiceleague = insere "Superman" "Clark Kent" (insere "Batman" "Bruce Wayne" []);;
```

Out[44]:
```
val justiceleague : (string * string) list =
  [("Superman", "Clark Kent"); ("Batman", "Bruce Wayne")]
```

In [45]:
```
let communaute =
    insere "Aragorn" "rodeur" (
        insere "Gandalf" "magicien" (
            insere "Gimli" "nain" (
                insere "Legolas" "elfe" (
                    insere "Frodon" "hobbit"
                    []
                )
            )
        )
    )
;;
```

Out[45]:
```
val communaute : (string * string) list =
  [("Aragorn", "rodeur"); ("Gandalf", "magicien"); ("Gimli", "nain");
   ("Legolas", "elfe"); ("Frodon", "hobbit")]
```

> La syntaxe est lourde, en comparaison d'un dictionnaire simple comme en Python...
>
> ```
> communaute = { "Aragorn": "rodeur", "Gandalf": "magicien", "Gimli": "nain",
> "Legolas": "elfe", "Frodon": "hobbit" }
> ```

Exercice 3 : `existe`

Première version, "à la main" :

In [46]:
```
let rec existe (cle : 'a) (l : ('a * 'b) list) : bool =
    match l with
    | [] -> false
    | (k, _) :: _ when cle = k -> true
    | _ :: q -> existe cle q
;;
```

Out[46]:
```
val existe : 'a → ('a * 'b) list → bool = <fun>
```

In [47]:
```
assert (existe "Frodon" communaute) ;;
assert (not (existe "Boromir" communaute));;
```

Out[47]:
```
- : unit = ()
```

Out[47]:
```
- : unit = ()
```

En utilisant la bibliothèque standard :

In [48]:
```
let existe (cle : 'a) (l : ('a * 'b) list) : bool =
    List.exists (fun (k, _) -> cle = k) l
;;
```

Out[48]:
```
val existe : 'a → ('a * 'b) list → bool = <fun>
```

In [49]:
```
assert (existe "Frodon" communaute) ;;
assert (not (existe "Boromir" communaute));;
```

Out[49]:
```
- : unit = ()
```

Out[49]:
```
- : unit = ()
```

In [50]:
```
let existe = List.mem_assoc;;
```

Out[50]:
```
val existe : 'a → ('a * 'b) list → bool = <fun>
```

In [51]:
```
assert (existe "Frodon" communaute) ;;
assert (not (existe "Boromir" communaute));;
```

Out[51]:
```
- : unit = ()
```

Out[51]:
```
- : unit = ()
```

Exercice 4 : `trouve`

On doit déclencher une erreur si la clé n'est pas trouvée. Pour être consistent, on déclenche la même que la fonction de la bibliothèque standard, `Not_found` :

In [52]:
```
List.assoc "ok" [];;
```

```
Exception: Not_found.
Raised at file "list.ml", line 158, characters 16-25
Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

In [53]:
```
let rec trouve (cle : 'a) (l : ('a * 'b) list) : 'b =
    match l with
    | [] -> raise Not_found
    | (k, v) :: _ when cle = k -> v
    | _ :: q -> trouve cle q
;;
```

Out[53]:
```
val trouve : 'a → ('a * 'b) list → 'b = <fun>
```

In [54]:
```
assert ((trouve "Gandalf" communaute) = "magicien");;
assert (try (trouve "Boromir" communaute) = "guerrier" with Not_found -> true);;
```

Out[54]:
```
- : unit = ()
```

Out[54]:
```
- : unit = ()
```

Avec la bibliothèque standard :

In [55]:
```
let trouve = List.assoc;;
```

Out[55]:
```
val trouve : 'a → ('a * 'b) list → 'b = <fun>
```

In [56]:
```
assert ((trouve "Gandalf" communaute) = "magicien");;
assert (try (trouve "Boromir" communaute) = "guerrier" with Not_found -> true);;
```

Out[56]:
```
- : unit = ()
```

Out[56]:
```
- : unit = ()
```

Exercice 5 : `supprime`

On choisit la sémantique suivante : l'exception `Not_found` est levée si la clé n'est pas présente. On supprime sinon la *première* occurence de la clé (rappel : `insere` ajoute `(cle, valeur)` même si `cle` est déjà présente).

In [57]:
```
let rec supprime (cle : 'a) (l : ('a*'b) list) : ('a*'b) list =
  match l with
  | [] -> raise Not_found
  | (k, _) :: q when cle = k -> q
  | p :: q -> p :: supprime cle q
;;
```

Out[57]:
```
val supprime : 'a → ('a * 'b) list → ('a * 'b) list = <fun>
```

Par exemple :

In [58]:
```
communaute;;
```

Out[58]:
```
- : (string * string) list =
[("Aragorn", "rodeur"); ("Gandalf", "magicien"); ("Gimli", "nain");
 ("Legolas", "elfe"); ("Frodon", "hobbit")]
```

In [59]:
```
supprime "Gandalf" [ ];;
```

```
Exception: Not_found.
Raised at file "[57]", line 3, characters 18-27
Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

In [60]:
```
let fin_film_1 = supprime "Gandalf" communaute;;
```

Out[60]:
```
val fin_film_1 : (string * string) list =
  [("Aragorn", "rodeur"); ("Gimli", "nain"); ("Legolas", "elfe");
   ("Frodon", "hobbit")]
```

In [61]:
```
let dans100ans = supprime "Frodon" communaute;;
```

Out[61]:
```
val dans100ans : (string * string) list =
  [("Aragorn", "rodeur"); ("Gandalf", "magicien"); ("Gimli", "nain");
   ("Legolas", "elfe")]
```

In [62]:
```
let debut_film_3 = insere "Gandalf" "magicien blanc" fin_film_1;;
```

Out[62]:
```
val debut_film_3 : (string * string) list =
  [("Gandalf", "magicien blanc"); ("Aragorn", "rodeur"); ("Gimli", "nain");
   ("Legolas", "elfe"); ("Frodon", "hobbit")]
```

Question bonus : avec des tables d'associations

La bibliothèque standard fournit le module Map (http://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.html#VALMake). Il faut au préalable créer le bon module (syntaxe un peu difficile, avec un *foncteur*).

In [63]:
```
module M = Map.Make ( struct
    type t = int
    let compare = compare
  end);;

let t : string M.t = (M.add 1 "1" (M.add 2 "2" (M.add 3 "3" M.empty)));;
```

Out[63]:
```
module M :
  sig
    type key = int
    type +'a t
    val empty : 'a t
    val is_empty : 'a t → bool
    val mem : key → 'a t → bool
    val add : key → 'a → 'a t → 'a t
    val singleton : key → 'a → 'a t
    val remove : key → 'a t → 'a t
    val merge :
      (key → 'a option → 'b option → 'c option) → 'a t → 'b t → 'c t
    val union : (key → 'a → 'a → 'a option) → 'a t → 'a t → 'a t
    val compare : ('a → 'a → int) → 'a t → 'a t → int
    val equal : ('a → 'a → bool) → 'a t → 'a t → bool
    val iter : (key → 'a → unit) → 'a t → unit
    val fold : (key → 'a → 'b → 'b) → 'a t → 'b → 'b
    val for_all : (key → 'a → bool) → 'a t → bool
    val exists : (key → 'a → bool) → 'a t → bool
    val filter : (key → 'a → bool) → 'a t → 'a t
    val partition : (key → 'a → bool) → 'a t → 'a t * 'a t
    val cardinal : 'a t → int
    val bindings : 'a t → (key * 'a) list
    val min_binding : 'a t → key * 'a
    val max_binding : 'a t → key * 'a
    val choose : 'a t → key * 'a
    val split : key → 'a t → 'a t * 'a option * 'a t
    val find : key → 'a t → 'a
    val map : ('a → 'b) → 'a t → 'b t
    val mapi : (key → 'a → 'b) → 'a t → 'b t
  end
```

Out[63]:
```
val t : string M.t = <abstr>
```

In [64]:
```
let _ = M.mem 1 t;;
let _ = M.mem 2 t;;
let _ = M.mem 4 t;;

let _ = M.find 1 t;;
let _ = M.find 2 t;;
let _ = M.find 4 t;;

let _ = M.remove 1 t;;
let _ = M.remove 2 t;;
let _ = M.remove 4 t;;
```

Out[64]:
```
- : bool = true
```

Out[64]:
```
- : bool = true
```

Out[64]:
```
- : bool = false
```

Out[64]:
```
- : string = "1"
```

Out[64]:
```
- : string = "2"
```

```
Exception: Not_found.
Raised at file "map.ml", line 122, characters 16-25
Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

Automates finis déterministes

Types de données

Les listes d'association sont utilises pour stocker les transitions : pour chaque état, on stocke une liste de règle associant une lettre lue à l'état d'arrivée de la transition.

In [65]:
```
type ('a, 'b) assoc = ('a * 'b) list;;
type lettre = A | B | C;;

type mot = lettre list;; (* [lettre array] marche aussi bien ! *)
type langage = mot list;;
type etat = int;;
```

Out[65]:
```
type ('a, 'b) assoc = ('a * 'b) list
```

Out[65]:
```
type lettre = A | B | C
```

Out[65]:
```
type mot = lettre list
```

Out[65]:
```
type langage = mot list
```

Out[65]:
```
type etat = int
```

In [66]:
```
(* Automate fini déterministe *)
type afd = {
   taille : int;
   initial : etat;
   finals : etat list;
   (* on peut aussi utiliser : *)
   (* transition : (etat, (lettre, etat) assoc) assoc; *) (* comme une fonction q -> a -> q' *)
   (* transition : ((etat, lettre), etat) assoc; *) (* comme une fonction (q, a) -> q' *)
   transition : (lettre, etat) assoc array
};;
```

Out[66]:
```
type afd = {
  taille : int;
  initial : etat;
  finals : etat list;
  transition : (lettre, etat) assoc array;
}
```

Affichage (PAS DANS LE TP)

On va utiliser le langage dot (https://graphviz.readthedocs.io/en/stable/manual.html#using-raw-dot) pour afficher facilement des graphes, et donc ici, des automates. Plutôt que d'utiliser une bibliothèque, on va écrire une fonction `dot` qui transforme un automate fini déterministe a en un fichier `out.dot` qui est ensuite converti en SVG (pour être affiché ici).

In [67]:
```
let string_of_lettre = function
   | A -> "A"
   | B -> "B"
   | C -> "C"
;;
```

```
File "[67]", line 2, characters 6-7:
Warning 41: A belongs to several types: lettre lettre
The first one was selected. Please disambiguate if this is wrong.
```

Out[67]:
```
val string_of_lettre : lettre → string = <fun>
```

In [68]:
```
let lettre_of_string = function
  | "A" -> A
  | "B" -> B
  | "C" -> C
  | _ -> failwith "Lettre pas dans Sigma"
;;
```

```
File "[68]", line 2, characters 13-14:
Warning 41: A belongs to several types: lettre lettre
The first one was selected. Please disambiguate if this is wrong.
```

Out[68]:
```
val lettre_of_string : string → lettre = <fun>
```

In [69]:
```
let dot (nom : string) (a : afd) : unit =
  let f = open_out nom in
  let print_edge i l = try
    let e = List.assoc l a.transition.(i) in
    Printf.fprintf f "  %d -> %d [label=%s]\n"
    i e (string_of_lettre l)
  with Not_found -> ()
  in
  Printf.fprintf f "digraph g {\n";
  Printf.fprintf f "  node [shape=circle];\n";
  for i = 0 to a.taille-1 do
    print_edge i A;
    print_edge i B;
    print_edge i C
  done;
  Printf.fprintf f "}\n";
  close_out f;
;;
```

Out[69]:
```
val dot : string → afd → unit = <fun>
```

Reconnaissance d'un mot

Une première approche est d'écrire une fonction récursive qui lit la première lettre du mot `m` et continue. On peut aussi écrire une fonction itérative qui boucle sur les lettres du mot `m`, et garde un `q : etat ref` pour l'état courant.

On peut utiliser les fonctions `trouve` et `existe` que l'on a écrit plus haut, ou bien utiliser `List.mem_assoc` et `List.assoc` de la bibliothèque standard, comme on veut.

Récursivement

In [70]:
```
let lecture (a : afd) (m : mot) : bool =
  let rec lire_lettre (e : etat) (m : mot) : bool =
    match m with
    | l::u ->
      if List.mem_assoc l a.transition.(e) then
        lire_lettre (List.assoc l a.transition.(e)) u
      else false
    | [] ->
      List.mem e a.finals
  in
  lire_lettre a.initial m
;;
```

Out[70]:
```
val lecture : afd → mot → bool = <fun>
```

Itérativement

In [74]:
```
let lecture2 (a : afd) (m : mot) : bool =
  let q = ref (a.initial) in
  List.iter (fun l -> begin
    if List.mem_assoc l a.transition.(!q) then
      q := List.assoc l a.transition.(!q)
    end
  ) m;
  List.mem !q a.finals;
;;
```

Out[74]:
```
val lecture2 : afd → mot → bool = <fun>
```

Deux exemples d'automates

In [75]:
```
let fin_ba = {
  taille = 3;
  initial = 0;
  finals = [2];
  (*transition = [ (* si ((etat * lettre) * etat) list *)
    ((0, A), 0); ((0, B), 1); ((0, C), 0));
    ((1, A), 2); ((1, B), 1); ((1, C), 0));
    ((2, A), 0); ((2, B), 1); ((2, C), 0));
  ]*)
  (*transition = [ (* si ((etat * (lettre * etat) list) list *)
    (0, [(A, 0); (B, 1); (C, 0)]);
    (1, [(A, 2); (B, 1); (C, 0)]);
    (2, [(A, 0); (B, 1); (C, 0)]);
  ])*)
  transition = [| (* si ((lettre, etat) list) array *)
    [(A, 0); (B, 1); (C, 0)]; (* état 0 *)
    [(A, 2); (B, 1); (C, 0)]; (* état 1 *)
    [(A, 0); (B, 1); (C, 0)]; (* état 1 *)
  |]
};;
```

Out[75]:
```
val fin_ba : afd =
  {taille = 3; initial = 0; finals = [2];
   transition =
    [|[(A, 0); (B, 1); (C, 0)]; [(A, 2); (B, 1); (C, 0)];
      [(A, 0); (B, 1); (C, 0)]|]}
```

In [76]:
```
dot "afd__fin_ba.dot" fin_ba;;
Sys.command "ls -larth afd__fin_ba.dot";;
Sys.command "cat afd__fin_ba.dot";;
```

Out[76]:
```
- : unit = ()
```

```
-rw-r--r-- 1 lilian lilian 208 nov.  27 15:08 afd__fin_ba.dot
```

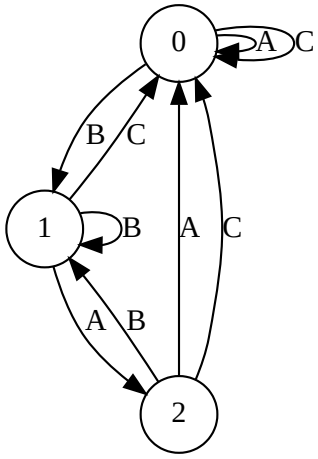
Out[76]:
```
- : int = 0
```

```
digraph g {
  node [shape=circle];
  0 → 0 [label=A]
  0 → 1 [label=B]
  0 → 0 [label=C]
  1 → 2 [label=A]
  1 → 1 [label=B]
  1 → 0 [label=C]
  2 → 0 [label=A]
  2 → 1 [label=B]
  2 → 0 [label=C]
}
```

Out[76]:
```
- : int = 0
```

In [77]:
```
Sys.command "dot -Tsvg -o afd__fin_ba.svg afd__fin_ba.dot";;
Sys.command "ls -larth afd__fin_ba.svg";;
```

Out[77]:
```
- : int = 0

-rw-r--r-- 1 lilian lilian 5,5K nov.  27 15:08 afd__fin_ba.svg

- : int = 0
```



Autre exemple :

In [78]:
```
let debut_ab = {
   taille = 3;
   initial = 0;
   finals = [2];
   transition = [|
      [(A, 1)];
      [(B, 2)];
      [(A, 2); (B, 2); (C, 2)]
   |]
};;
```

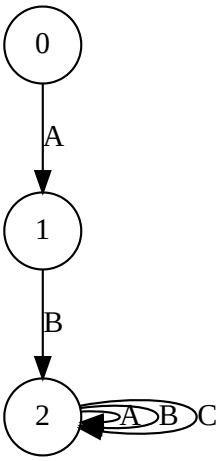
Out[78]:
```
val debut_ab : afd =
  {taille = 3; initial = 0; finals = [2];
   transition = [|[(A, 1)]; [(B, 2)]; [(A, 2); (B, 2); (C, 2)]|]}
```

In [79]:
```
dot "afd__debut_ab.dot" debut_ab;;
Sys.command "ls -larth afd__debut_ab.dot";;
Sys.command "cat afd__debut_ab.dot";;
```

Out[79]:
```
- : unit = ()

-rw-r--r-- 1 lilian lilian 132 nov.  27 15:08 afd__debut_ab.dot

- : int = 0

digraph g {
  node [shape=circle];
  0 → 1 [label=A]
  1 → 2 [label=B]
  2 → 2 [label=A]
  2 → 2 [label=B]
  2 → 2 [label=C]
}

- : int = 0
```

In [80]:
```
Sys.command "dot -Tsvg -o afd__debut_ab.svg afd__debut_ab.dot";;
Sys.command "ls -larth afd__debut_ab.svg";;
```

```
Out[80]: - : int = 0

         -rw-rw-r-- 1 lilian lilian 3,5K nov.  27 15:08 afd__debut_ab.svg

Out[80]: - : int = 0
```



Exemple de lectures

On doit vérifier que ces deux automates reconnaissent bien respectivement les mots terminants par *ba* et les mots commençants par *ab*.

In [81]:
```
let _ = lecture  fin_ba [A;B;A];;
let _ = lecture  fin_ba [A;B;A;A];;

let _ = lecture  debut_ab [A;B;A];;
let _ = lecture  debut_ab [B;A;A];;
```

```
Out[81]: - : bool = true

Out[81]: - : bool = false

Out[81]: - : bool = true

Out[81]: - : bool = false
```

In [82]:
```
let _ = lecture2  fin_ba [A;B;A];;
let _ = lecture2  fin_ba [A;B;A;A];;

let _ = lecture2  debut_ab [A;B;A];;
let _ = lecture2  debut_ab [B;A;A];;
```

```
Out[82]: - : bool = true

Out[82]: - : bool = false

Out[82]: - : bool = true

Out[82]: - : bool = false
```

Complétion (DIFFICILE)

In [40]:
```
let complete (a:afd) : afd =
  let puit = a.taille in
  let ajoute_arc (l : lettre) (e : etat) (asso : (lettre, etat) assoc) =
    if List.mem_assoc l a.transition.(e)
    then asso
    else (l, puit) :: asso
  in
  let complete_etat e =
    if e < a.taille then
      ajoute_arc A e
      (ajoute_arc B e
        (ajoute_arc C e
          a.transition.(e)
        )
      )
    else
      [(A, puit); (B, puit); (C, puit)]
  in
  {
    a with
    taille = a.taille + 1;
    transition = Array.init (a.taille + 1) complete_etat
  }
;;
```

Out[40]:
```
val complete : afd → afd = <fun>
```

In [41]:
```
let com_debut_ab = complete debut_ab;;
```

Out[41]:
```
val com_debut_ab : afd =
  {taille = 4; initial = 0; finals = [2];
   transition =
    [|[(B, 3); (C, 3); (A, 1)]; [(A, 3); (C, 3); (B, 2)];
      [(A, 2); (B, 2); (C, 2)]; [(A, 3); (B, 3); (C, 3)]|]}
```

In [42]:
```
dot "afd__com_debut_ab.dot" com_debut_ab;;
Sys.command "ls -larth afd__com_debut_ab.dot";;
Sys.command "cat afd__com_debut_ab.dot";;
```

Out[42]:
```
- : unit = ()
```

```
-rw-rw-r-- 1 lilian lilian 265 oct.  10 17:23 afd__com_debut_ab.dot
```

Out[42]:
```
- : int = 0
```

```
digraph g {
  node [shape=circle];
  0 → 1 [label=A]
  0 → 3 [label=B]
  0 → 3 [label=C]
  1 → 3 [label=A]
  1 → 2 [label=B]
  1 → 3 [label=C]
  2 → 2 [label=A]
  2 → 2 [label=B]
  2 → 2 [label=C]
  3 → 3 [label=A]
  3 → 3 [label=B]
  3 → 3 [label=C]
}
```

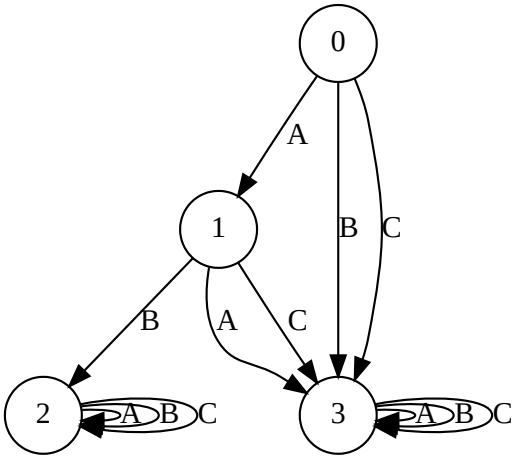
Out[42]:
```
- : int = 0
```

In [43]:
```
Sys.command "dot -Tsvg -o afd__com_debut_ab.svg afd__com_debut_ab.dot";;
Sys.command "ls -larth afd__com_debut_ab.svg";;
```

Out[43]:
```
- : int = 0
```

```
-rw-rw-r-- 1 lilian lilian 6,6K oct.  10 17:23 afd__com_debut_ab.svg
```

Out[43]:
```
- : int = 0
```



Complémentaire (plus dur)

In [52]:
```
let complementaire (a : afd) : afd =
  let rec finals = function
    | n when n < 0 -> []
    | n when n != a.initial -> n :: finals (n-1)
    | n -> finals (n-1)
  in
  let a' = complete a in
  {
    taille = a.taille +1;
    initial = a.initial;
    finals = finals (a.taille + 1);
    transition = a'.transition
  }
```

Out[52]:
```
val complementaire : afd → afd = <fun>
```

In [53]:
```
let not_debut_ab = complementaire debut_ab;;
```

Out[53]:
```
val not_debut_ab : afd =
  {taille = 4; initial = 0; finals = [4; 3; 2; 1];
   transition =
    [|[(B, 3); (C, 3); (A, 1)]; [(A, 3); (C, 3); (B, 2)];
      [(A, 2); (B, 2); (C, 2)]; [(A, 3); (B, 3); (C, 3)]|]}
```

In [55]:
```
dot "afd__not_debut_ab.dot" not_debut_ab;;
Sys.command "ls -larth afd__not_debut_ab.dot";;
Sys.command "cat afd__not_debut_ab.dot";;
```

Out[55]:
```
- : unit = ()
```

```
-rw-rw-r-- 1 lilian lilian 265 oct.  10 17:39 afd__not_debut_ab.dot
```

Out[55]:
```
- : int = 0
```

```
digraph g {
  node [shape=circle];
  0 → 1 [label=A]
  0 → 3 [label=B]
  0 → 3 [label=C]
  1 → 3 [label=A]
  1 → 2 [label=B]
  1 → 3 [label=C]
  2 → 2 [label=A]
  2 → 2 [label=B]
  2 → 2 [label=C]
  3 → 3 [label=A]
  3 → 3 [label=B]
  3 → 3 [label=C]
}
```

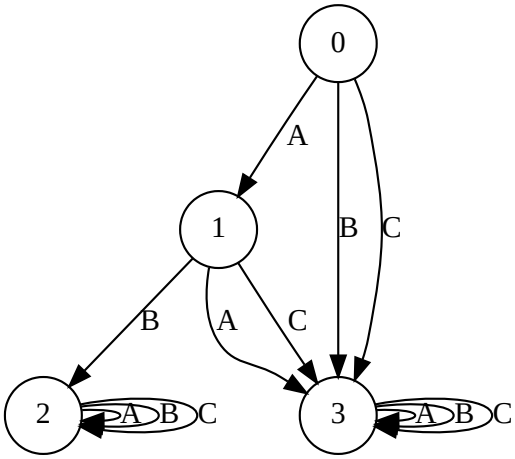
Out[55]:
```
- : int = 0
```

In [56]:
```
Sys.command "dot -Tsvg -o afd__not_debut_ab.svg afd__not_debut_ab.dot";;
Sys.command "ls -larth afd__not_debut_ab.svg";;
```

Out[56]:
```
- : int = 0
```

```
-rw-rw-r-- 1 lilian lilian 6,6K oct.  10 17:43 afd__not_debut_ab.svg
```

Out[56]:
```
- : int = 0
```



Expressions régulières

On se fixe $\Sigma = \{a, b, c\}$.

On rappelle la grammaire des expressions régulières :

```
<exp> ::=
| ∅
| ε
| a (lettre dans Sigma)
| <exp> + <exp>
| <exp> . <exp>
| <exp>*
```

Exercice 10 : `regexp`

On représente ça le plus simplement possible, avec un type multiple :

In [84]:
```
type regexp =
  | Vide
  | Epsilon (* On peut faire sans ! *)
  | Lettre of lettre
  | Somme of (regexp * regexp)
  | Concat of (regexp * regexp)
  | Etoile of regexp
```

Out[84]:
```
type regexp =
    Vide
  | Epsilon
  | Lettre of lettre
  | Somme of (regexp * regexp)
  | Concat of (regexp * regexp)
  | Etoile of regexp
```

Exercice 11 : deux regexp pour les deux automates A_1, A_2

On peut définir des valeurs intermédiaires pour écrire les exemples plus rapidement :

In [85]:
```
let a = Lettre A;;
let b = Lettre B;;
let c = Lettre C;;
```

Out[85]:
```
val a : regexp = Lettre A
```

Out[85]:
```
val b : regexp = Lettre B
```

Out[85]:
```
val c : regexp = Lettre C
```

In [86]:
```
let sigma = Somme (Somme (a, b), c);;
let sigmaetoile = Etoile sigma;;

let la1 = Concat (sigmaetoile, Concat (a,b));;
let la2 = Concat (Concat (b, a), sigmaetoile);;
```

```
Out[86]: val sigma : regexp = Somme (Somme (Lettre A, Lettre B), Lettre C)

Out[86]: val sigmaetoile : regexp =
           Etoile (Somme (Somme (Lettre A, Lettre B), Lettre C))

Out[86]: val la1 : regexp =
           Concat
            (Etoile (Somme (Somme (Lettre A, Lettre B), Lettre C)),
             Concat (Lettre A, Lettre B))

Out[86]: val la2 : regexp =
           Concat
            (Concat (Lettre B, Lettre A),
             Etoile (Somme (Somme (Lettre A, Lettre B), Lettre C)))
```

Un exemple plus long sera l'expression régulière reconnaissant $\Sigma^7\Sigma^*$ les mots de longueur au moins *[Math Processing Error]*.

In [88]:
```
let rec au_moins_longueur = function
  | 0 -> sigmaetoile
  | n -> Concat (sigma, au_moins_longueur (n - 1))
;;

let au_moins7 = au_moins_longueur 7;;
```

```
Out[88]: val au_moins_longueur : int → regexp = <fun>

Out[88]: val au_moins7 : regexp =
           Concat
            (Somme (Somme (Lettre A, Lettre B), Lettre C),
             Concat
              (Somme (Somme (Lettre A, Lettre B), Lettre C),
               Concat
                (Somme (Somme (Lettre A, Lettre B), Lettre C),
                 Concat
                  (Somme (Somme (Lettre A, Lettre B), Lettre C),
                   Concat
                    (Somme (Somme (Lettre A, Lettre B), Lettre C),
                     Concat
                      (Somme (Somme (Lettre A, Lettre B), Lettre C),
                       Concat
                        (Somme (Somme (Lettre A, Lettre B), Lettre C),
                         Concat
                          (Somme (Somme (Lettre A, Lettre B), Lettre C),
                           Etoile (Somme (Somme (Lettre A, Lettre B), Lettre C)))))))))
```

Exercice 12 : `to_string`

On peut faire une première version assez simple, qui sera assez moche puisqu'il y aura plein de parenthèses partout :

In [89]:
```
let rec regexp_to_string = function
  | Vide -> "{}"
  | Epsilon -> "Epsilon"
  | Lettre A -> "A"
  | Lettre B -> "B"
  | Lettre C -> "C"
  | Somme (r1, r2) ->
     "(" ^ (regexp_to_string r1) ^ " + " ^ (regexp_to_string r2) ^ ")"
  | Concat (r1, r2) ->
     "(" ^ (regexp_to_string r1) ^ " . " ^ (regexp_to_string r2) ^ ")"
  | Etoile r -> "(" ^ (regexp_to_string r) ^ ")*"
;;
```

```
Out[89]: val regexp_to_string : regexp → string = <fun>
```

In [64]:
```
let _ = regexp_to_string la1;;
let _ = regexp_to_string la2;;
let _ = regexp_to_string au_moins7;;
```

Out[64]:
```
- : string = "((((A + B) + C))* . (A . B))"
```

Out[64]:
```
- : string = "((B . A) . (((A + B) + C))*)"
```

Out[64]:
```
- : string =
"(((A + B) + C) . (((A + B) + C) . (((A + B) + C) . (((A + B) + C) . (((A + B) + C)
. (((A + B) + C) . (((A + B) + C) . (((A + B) + C))*)))))))"
```

On peut chercher à faire un peu plus joli. L'argument `last` garde en mémoire le dernier symbole binaire ou unaire lu, `Somme`, `Concat` ou `Etoile`. Cela permet de ne pas mettre des parenthèses quand on affiche `(A+B+C)` au lieu de `(A+(B+C))` et `(A.B.C)` au lieu de `(A.(B.C))`.

In [90]:
```
open Printf;;

let rec to_string last = function
    | Vide -> "{}"
    | Epsilon -> "Epsilon"
    | Lettre A -> "A"
    | Lettre B -> "B"
    | Lettre C -> "C"
    | Somme (r1, r2) ->
        if last="+" || last="*" then
            sprintf "%s + %s" (to_string "+" r1) (to_string "+" r2)
        else
            sprintf "(%s + %s)" (to_string "+" r1) (to_string "+" r2)
    | Concat (r1, r2) ->
        if last="." || last="*" then
            sprintf "%s . %s" (to_string "." r1) (to_string "." r2)
        else
            sprintf "(%s . %s)" (to_string "." r1) (to_string "." r2)
    | Etoile r -> sprintf "(%s)*" (to_string "*" r)
;;

let regexp_to_string = to_string "*";;
```

Out[90]:
```
val to_string : string → regexp → string = <fun>
```

Out[90]:
```
val regexp_to_string : regexp → string = <fun>
```

Exemples :

In [91]:
```
let _ = regexp_to_string Vide;;
```

Out[91]:
```
- : string = "{}"
```

In [92]:
```
let _ = regexp_to_string Epsilon;;
```

Out[92]:
```
- : string = "Epsilon"
```

In [93]:
```
let _ = regexp_to_string (Etoile Epsilon);;
```

Out[93]:
```
- : string = "(Epsilon)*"
```

In [94]:
```
let _ = regexp_to_string la1;;
let _ = regexp_to_string la2;;
let _ = regexp_to_string au_moins7;;
```

Out[94]:
```
- : string = "(A + B + C)* . A . B"
```

Out[94]:
```
- : string = "B . A . (A + B + C)*"
```

Out[94]:
```
- : string =
"(A + B + C) . (A + B + C) . (A + B + C) . (A + B + C) . (A + B + C) . (A + B + C)
. (A + B + C) . (A + B + C)*"
```

Exercice 13 : `est_vide`

On teste si le langage généré par l'expression régulière est vide ou non. Une étoile n'est jamais vide, même $\varepsilon^* = \emptyset^* = \{\varepsilon\}$.

In [102]:
```
let rec est_vide = function
    | Vide -> true
    | Epsilon -> false
    | Lettre _ -> false
    | Somme (r1, r2) | Concat (r1, r2) -> est_vide r1 && est_vide r2
    | Etoile _ -> false (* piège ! *)
;;
```

Out[102]:
```
val est_vide : regexp → bool = <fun>
```

In [103]:
```
let _ = est_vide Vide;;
let _ = est_vide sigma;;
let _ = est_vide la1;;
let _ = est_vide la2;;
```

Out[103]:
```
- : bool = true
```

Out[103]:
```
- : bool = false
```

Out[103]:
```
- : bool = false
```

Out[103]:
```
- : bool = false
```

In [104]:
```
let _ = est_vide (Etoile Vide);;
let _ = est_vide (Etoile Epsilon);;
let _ = est_vide Epsilon;;
```

Out[104]:
```
- : bool = false
```

Out[104]:
```
- : bool = false
```

Out[104]:
```
- : bool = false
```

Exercice 14 : `est_fini`

Pour tester si le langage généré est fini, il faut réfléchir un peu plus, parce qu'une étoile e^* est infinie à condition que le langage généré par l'expression e soit non vide **et pas réduit au sigleton** $\{\varepsilon\}$!

In [106]:
```
let rec est_vide_ou_epsilon = function
    | Vide -> true
    | Epsilon -> true
    | Lettre _ -> false
    | Somme (r1, r2) | Concat (r1, r2) -> est_vide_ou_epsilon r1 || est_vide_ou_epsilon r2
    | Etoile r -> est_vide_ou_epsilon r
;;
```

Out[106]:
```
val est_vide_ou_epsilon : regexp → bool = <fun>
```

In [107]:
```
let rec est_fini = function
    | Vide -> true
    | Epsilon -> true
    | Lettre _ -> true
    | Somme (r1, r2) | Concat (r1, r2) -> est_fini r1 && est_fini r2
    | Etoile r -> est_vide_ou_epsilon r
    (* Piège car [Etoile Vide] est fini, [Etoile Epsilon] est fini aussi ! *)
;;
```

Out[107]:
```
val est_fini : regexp → bool = <fun>
```

In [108]:
```
let _ = est_fini Vide;;
let _ = est_fini Epsilon;;
let _ = est_fini sigma;;
let _ = est_fini la1;;
let _ = est_fini la2;;
```

```
Out[108]: - : bool = true

Out[108]: - : bool = true

Out[108]: - : bool = false

Out[108]: - : bool = false
```

In [110]:
```
let _ = est_fini (Etoile Vide);;
let _ = est_fini (Etoile Epsilon);;
let _ = est_fini (Etoile (Somme (Epsilon, Epsilon)));;
let _ = est_fini (Etoile (Somme (Vide, Epsilon)));;
let _ = est_fini (Etoile (Somme (Vide, Vide)));;
let _ = est_fini (Etoile (Concat (Epsilon, Epsilon)));;
let _ = est_fini (Etoile (Concat (Vide, Epsilon)));;
let _ = est_fini (Etoile (Concat (Vide, Vide)));;
let _ = est_fini (Etoile sigma);;
```

```
Out[110]: - : bool = true

Out[110]: - : bool = true

Out[110]: - : bool = true

Out[110]: - : bool = true

Out[110]: - : bool = true

Out[110]: - : bool = true

Out[110]: - : bool = true

Out[110]: - : bool = true

Out[110]: - : bool = false
```

Exercice 15 : `pile_ou_face`

On pense bien à initialiser le générateur de nombres pseudo aléatoires avec `Random.self_init` (https://caml.inria.fr/pub/docs/manual-ocaml/libref/Random.html#VALself_init).

In [111]:
```
type piece = Pile | Face;;
Random.self_init ();;

let pile_ou_face () =
  match Random.int 2 with
  | 0 -> Pile
  | 1 -> Face
  | _ -> failwith "impossible"
;;
```

```
Out[111]: type piece = Pile | Face

Out[111]: - : unit = ()

Out[111]: val pile_ou_face : unit → piece = <fun>
```

Par exemple :

In [113]:
```
let _ = Array.init 10 (fun _ -> pile_ou_face ());;
```

```
Out[113]: - : piece array =
          [|Pile; Pile; Pile; Pile; Pile; Pile; Face; Face; Pile; Face|]
```

In [114]:
```
let _ = Array.init 10 (fun _ -> pile_ou_face ());;
```

Out[114]:
```
- : piece array =
[|Face; Face; Face; Pile; Pile; Face; Pile; Face; Pile; Pile|]
```

In [115]:
```
let _ = Array.init 10 (fun _ -> pile_ou_face ());;
```

Out[115]:
```
- : piece array =
[|Face; Face; Pile; Pile; Pile; Face; Pile; Pile; Pile; Face|]
```

Exercice 16 : `mot_aleatoire`

Ce n'est pas trop compliqué : l'aléatoire est utilisé dans une somme, pour choisir l'un ou l'autre des expressions avec probabilité $1/2$, et dans une étoile.

En fait, il faut faire attention avec ces deux cas, parce que si l'un des deux morceaux est vide, il faut choisir l'autre (donc `est_fini` sera utile).

A noter que le choix d'implémentation de l'aléatoire dans l'étoile donne une distribution sur la longueur qui est non triviale. Un bon exercice serait de trouver la distribution de la longueur d'un mot aléatoire généré par la fonction ci-dessous à partir de l'expression régulière a^*. (est-ce toujours 2 ? une variable aléatoire suivant une loi de Poisson de paramètre $\lambda = 1/2$? une loi exponentielle ?). Envoyez moi vos réponsez par mail (http://perso.crans.org/besson/callme) (ou ce formulaire (http://perso.crans.org/besson/contact/)).

In [116]:
```
let rec mot_aleatoire = function
    | Vide -> failwith "langage vide"
    | Epsilon -> [] (* mot vide = liste de lettres vides *)
    | Lettre l -> [l]
    (* si une est vide on doit pas la choisir *)
    | Somme (r1, r2) when est_vide r1 -> mot_aleatoire r2
    | Somme (r1, r2) when est_vide r2 -> mot_aleatoire r1
    | Somme (r1, r2) -> begin
        match pile_ou_face() with
        | Pile -> mot_aleatoire r1
        | Face -> mot_aleatoire r2
    end
    | Concat (r1, r2) ->
        let m1 = mot_aleatoire r1 in
        let m2 = mot_aleatoire r2 in
        m1 @ m2
    (* Etoile (quelque chose qui est vide) devrait marcher et renvoyer vide *)
    | Etoile r when est_vide r -> [] (* mot vide *)
    | Etoile r -> begin
        match pile_ou_face() with
        | Pile -> []
        | Face -> (mot_aleatoire r) @ (mot_aleatoire (Etoile r))
    end
;;
```

Out[116]:
```
val mot_aleatoire : regexp → lettre list = <fun>
```

On peut faire quelques exemples :

In [117]:
```
let _ = mot_aleatoire la1;;
let _ = mot_aleatoire la1;;
let _ = mot_aleatoire la1;;
let _ = mot_aleatoire la1;;
let _ = mot_aleatoire la1;;
let _ = mot_aleatoire la1;;
let _ = mot_aleatoire la1;;
```

```
Out[117]: - : lettre list = [A; B]
Out[117]: - : lettre list = [A; A; B; A; C; A; B]
Out[117]: - : lettre list = [C; A; B]
Out[117]: - : lettre list = [A; B]
Out[117]: - : lettre list = [A; C; A; B]
Out[117]: - : lettre list = [A; B]
Out[117]: - : lettre list = [A; A; B]
```

In [118]:
```
let _ = mot_aleatoire la2;;
let _ = mot_aleatoire la2;;
let _ = mot_aleatoire la2;;
let _ = mot_aleatoire la2;;
let _ = mot_aleatoire la2;;
let _ = mot_aleatoire la2;;
let _ = mot_aleatoire la2;;
```

```
Out[118]: - : lettre list = [B; A; B]
Out[118]: - : lettre list = [B; A; C; C]
Out[118]: - : lettre list = [B; A]
Out[118]: - : lettre list = [B; A; A]
Out[118]: - : lettre list = [B; A]
Out[118]: - : lettre list = [B; A]
Out[118]: - : lettre list = [B; A]
```

In [119]:
```
let _ = mot_aleatoire au_moins7;;
let _ = mot_aleatoire au_moins7;;
let _ = mot_aleatoire au_moins7;;
let _ = mot_aleatoire au_moins7;;
let _ = mot_aleatoire au_moins7;;
let _ = mot_aleatoire au_moins7;;
let _ = mot_aleatoire au_moins7;;
```

```
Out[119]: - : lettre list = [C; B; A; C; B; C; B]
Out[119]: - : lettre list = [C; C; C; C; B; C; C; C; C]
Out[119]: - : lettre list = [B; C; C; B; A; C; B]
Out[119]: - : lettre list = [B; B; A; C; A; B; B; B; B]
Out[119]: - : lettre list = [B; A; B; A; A; C; A; B; B]
Out[119]: - : lettre list = [C; A; A; C; A; B; B; C]
Out[119]: - : lettre list = [C; C; C; A; A; C; C]
```

Ici, on pourrait faire des expériences numériques pour afficher une distribution (empirique) sur la longueur des mots générés pour une certaine expression régulière.

> Note : le mot "généré" s'applique plutôt à une grammaire, on dit généralement "reconnu" par une expression régulière et un automate. Mais cette fonction `mot_aleatoire` permet bien, elle, de générer des mots.

Calcul de $\Sigma^k \cap L(A)$

Exercice 17 : `produit_cartesien`

C'est assez simple à faire, quand on ne s'embête pas à chercher à être très efficace (sur les concaténations). Par contre, cette implémentation est efficace sur les appels récursifs, elle utilise cette fonction interne `aux` et un accumulateur `acc`.

Notez l'implémentation générique qui permet de transformer comme on veut couple d'éléments des deux listes, de type `'a` et `'b`, en un élément de type `'c`. En pratique, `fun a b → (a, b)` sera utilisé pour faire le "vrai" produit cartésien.

In [14]:
```
let produit_cartesien (prod : 'a -> 'b -> 'c) (a : 'a list) (b : 'b list) : 'c list =
  let rec aux acc a =
  match a with
  | [] -> acc
  | va :: qa -> aux ((List.map (fun vb -> prod va vb) b) @ acc) qa
  in
  List.rev (aux [] a)
;;
```

Out[14]:
```
val produit_cartesien : ('a → 'b → 'c) → 'a list → 'b list → 'c list =
  <fun>
```

Par exemple :

In [15]:
```
produit_cartesien (fun a b -> (a, b)) [1; 2] ["ok"; "pas"; "probleme"];;
```

Out[15]:
```
- : (int * string) list =
[(1, "probleme"); (1, "pas"); (1, "ok"); (2, "probleme"); (2, "pas");
 (2, "ok")]
```

Liste de tous les mots de Σ^k

On peut commencer par construire Σ^k comme une expression régulière, c'est très simple, mais ça ne sera pas suffisant :

In [121]:
```
let rec sigma_k (k : int) : regexp =
  match k with
  | n when n < 1 -> Vide
  | 1 -> sigma
  | n -> Concat (sigma, sigma_k (n - 1))
;;
```

Out[121]:
```
val sigma_k : int → regexp = <fun>
```

In [122]:
```
regexp_to_string (sigma_k 0);;
regexp_to_string (sigma_k 1);;
regexp_to_string (sigma_k 4);;
regexp_to_string (sigma_k 12);;
```

Out[122]:
```
- : string = "{}"
```

Out[122]:
```
- : string = "A + B + C"
```

Out[122]:
```
- : string = "(A + B + C) . (A + B + C) . (A + B + C) . (A + B + C)"
```

Out[122]:
```
- : string =
"(A + B + C) . (A + B + C) . (A + B + C) . (A + B + C) . (A + B + C) . (A + B + C)
. (A + B + C) . (A + B + C) . (A + B + C) . (A + B + C) . (A + B + C) . (A + B +
C)"
```

On a besoin de créer une liste de mots, tous les mots dans Σ^k (il y en a exactement $|\Sigma|^k$, attention ça grandit vite !)

In [123]:
```
let alphabet = [A; B; C];; (* Sigma *)

let rec tous_mots_sigma_k (alphabet : lettre list) (k : int) : mot list =
  match k with
  | k when k < 1 -> []
  | 1 -> List.map (fun lettre -> [lettre]) alphabet
  | k -> List.concat (
    List.map (
      fun lettre -> (
        List.map (fun mot -> lettre :: mot)
        (tous_mots_sigma_k alphabet (k - 1))
      )
    )
    alphabet
  )
;;
```

```
File "[123]", line 1, characters 16-17:
Warning 41: A belongs to several types: lettre lettre
The first one was selected. Please disambiguate if this is wrong.
```

Out[123]:
```
val alphabet : lettre list = [A; B; C]
```

Out[123]:
```
val tous_mots_sigma_k : lettre list → int → mot list = <fun>
```

In [124]:
```
let _ = tous_mots_sigma_k alphabet 0;;
let _ = tous_mots_sigma_k alphabet 1;;
let _ = tous_mots_sigma_k alphabet 2;;
let _ = tous_mots_sigma_k alphabet 3;;
```

Out[124]:
```
- : mot list = []
```

Out[124]:
```
- : mot list = [[A]; [B]; [C]]
```

Out[124]:
```
- : mot list =
[[A; A]; [A; B]; [A; C]; [B; A]; [B; B]; [B; C]; [C; A]; [C; B]; [C; C]]
```

Out[124]:
```
- : mot list =
[[A; A; A]; [A; A; B]; [A; A; C]; [A; B; A]; [A; B; B]; [A; B; C]; [A; C; A];
 [A; C; B]; [A; C; C]; [B; A; A]; [B; A; B]; [B; A; C]; [B; B; A]; [B; B; B];
 [B; B; C]; [B; C; A]; [B; C; B]; [B; C; C]; [C; A; A]; [C; A; B]; [C; A; C];
 [C; B; A]; [C; B; B]; [C; B; C]; [C; C; A]; [C; C; B]; [C; C; C]]
```

Exercice 19 : `filtre`

C'est très rapide, et c'est exactement la fonction `List.filter` de la bibliothèque standard. Attention, en français c'est filtre (tre) et en anglais (américain) c'est filter (ter).

In [126]:
```
let rec filtre (pred : 'a -> bool) (l : 'a list) : 'a list =
  match l with
  | [] -> []
  | h :: q when pred h -> h :: (filtre pred q)
  | _ :: q -> filtre pred q
;;
```

Out[126]:
```
val filtre : ('a → bool) → 'a list → 'a list = <fun>
```

In [127]:
```
List.filter;;
```

Out[127]:
```
- : ('a → bool) → 'a list → 'a list = <fun>
```

In [97]:
```
filtre (fun x -> x mod 2 = 0) [1; 2; 3; 4];;
```

Out[97]:
```
- : int list = [2; 4]
```

```
In [128]: List.filter (fun x -> x mod 2 = 0) [1; 2; 3; 4];;
Out[128]: - : int list = [2; 4]
```

Exercice 20

C'est très facile ! Il suffit d'utiliser la fonction `lecture` comme un prédicat binaire :

```
In [98]: lecture;;
Out[98]: - : afd → mot → bool = <fun>
```

```
In [124]: let sigmak_inter_LA (k : int) (a : afd) : mot list =
              let s_k = tous_mots_sigma_k alphabet k in
              filtre (fun mot -> lecture a mot) s_k
          ;;
Out[124]: val sigmak_inter_LA : int → afd → mot list = <fun>
```

Exemples pour les deux automates du début tels que $L(A)$ soient $\Sigma^* ba$ et $ab\Sigma^*$. Il y a $|\Sigma|^2 = 3^2 = 9$ mots dans les deux cas, puisque 2 lettres parmi les 4 (pour des mots de Σ^4) sont déjà fixées.

```
In [126]: let _ = sigmak_inter_LA 4 fin_ba;;

          let _ = sigmak_inter_LA 4 debut_ab;;
Out[126]: - : mot list =
          [[A; A; B; A]; [A; B; B; A]; [A; C; B; A]; [B; A; B; A]; [B; B; B; A];
           [B; C; B; A]; [C; A; B; A]; [C; B; B; A]; [C; C; B; A]]
Out[126]: - : mot list =
          [[A; B; A; A]; [A; B; A; B]; [A; B; A; C]; [A; B; B; A]; [A; B; B; B];
           [A; B; B; C]; [A; B; C; A]; [A; B; C; B]; [A; B; C; C]]
```

Automate produit (PLUS DUR)

C'est plus dur mais assez guidé.

Exercice 21 : `bijection`

```
In [8]: type f_intint_int = (int * int -> int);;
        type f_int_intint = (int -> int * int);;
Out[8]: type f_intint_int = int * int → int
Out[8]: type f_int_intint = int → int * int
```

```
In [9]: let bijection (p : int) (q : int) : f_intint_int * f_int_intint =
            let f (n, m) = m + n * q in
            let finv x =
                let m = x mod q and n = x / q in
                assert ((f (n, m)) = x);
                (n, m);
            in
            f, finv
        ;;
        File "[9]", line 1, characters 15-16:
        Warning 27: unused variable p.
Out[9]: val bijection : int → int → f_intint_int * f_int_intint = <fun>
```

Il faut absolument la tester, au moins vérifier que $f^{-1}(f(n,m)) = (n,m)$ et $f(f^{-1}(x)) = x$ pour tout $n,m \in [0,p-1] \times [0,q-1]$et $x \in [0,pq-1]$.

In [10]:
```
let p = 2 and q = 4;;
let f, finv = bijection 2 4;;

for n = 0 to p - 1 do
   flush_all();
   for m = 0 to q - 1 do
      Printf.printf "\n%i, %i -> %i" n m (f (n, m));
      assert ((n, m) = finv (f (n, m)));
   done;
   flush_all();
done;;

for x = 0 to p*q - 1 do
   flush_all();
   let n, m = finv x in
   Printf.printf "\n%i -> %i, %i" x n m ;
   assert (x = f (finv x));
done;;
```

Out[10]:
```
val p : int = 2
val q : int = 4
```

Out[10]:
```
val f : f_intint_int = <fun>
val finv : f_int_intint = <fun>

0, 0 → 0
0, 1 → 1
0, 2 → 2
0, 3 → 3
1, 0 → 4
1, 1 → 5
1, 2 → 6
```

Out[10]:
```
- : unit = ()

1, 3 → 7
0 → 0, 0
1 → 0, 1
2 → 0, 2
3 → 0, 3
4 → 1, 0
5 → 1, 1
```

Out[10]:
```
- : unit = ()
```

Exercice 22

On utilise `produit_cartesien` pour les états finaux, une simple paire pour l'état initial, et un peu de calcul pour les transitions. L'idée est d'utiliser cette bijection f pour coder les paires comme des entiers simples (et donc produire un automate représenté par un `afd`).

In [18]:
```
let alphabet = [A; B; C];;

let automate_produit (a1 : afd) (a2 : afd) =
   let p, i1, f1, d1 = a1.taille, a1.initial, a1.finals, a1.transition in
   let q, i2, f2, d2 = a2.taille, a2.initial, a2.finals, a2.transition in
   (* les bijections *)
   let taille = p * q in
   let f, finv = bijection p q in
   (* état initial *)
   let initial = f (i1, i2) in
   (* peut contenir des doublons, pas grave *)
   let finals = List.map f (produit_cartesien (fun x y -> (x, y)) f1 f2) in
   (* et moins trivial pour les transitions *)
   let transition = Array.init taille (fun ij -> (* pour l'état (i, j) *)
      let i, j = finv ij in
      (* d1.(i) est une liste de (lettre, etat) = (a, q1) pour i --a-> q1 *)
      let transition_i_1 = d1.(i) in
      (* d2.(j) est une liste de (lettre, etat) = (b, q2) pour j --b-> q2 *)
      let transition_j_2 = d2.(j) in
      (* on doit trouver les transitions avec la meme lettre et produire i --a-> f q1 q2 *)
      List.concat (
         List.map (fun lettre ->
            (* pour cette lettre on cherche la transition jointe qui convient, si elle existe *)
            if (List.mem_assoc lettre transition_i_1) && (List.mem_assoc lettre transition_j_2) then
            begin
               let q1 = List.assoc lettre transition_i_1 in
               let q2 = List.assoc lettre transition_j_2 in
               [(lettre, f(q1, q2))]
            end else []
         )
      alphabet)
   ) in
   { taille; initial; finals; transition }
;;
```

Out[18]:
```
val alphabet : lettre list = [A; B; C]
```

Out[18]:
```
val automate_produit : afd → afd → afd = <fun>
```

Exemple :

In [11]:
```
debut_ab;;
fin_ba;;
```

Out[11]:
```
- : afd =
{taille = 3; initial = 0; finals = [2];
 transition = [|[(A, 1)]; [(B, 2)]; [(A, 2); (B, 2); (C, 2)]|]}
```

Out[11]:
```
- : afd =
{taille = 3; initial = 0; finals = [2];
 transition =
  [|[(A, 0); (B, 1); (C, 0)]; [(A, 2); (B, 1); (C, 0)];
    [(A, 0); (B, 1); (C, 0)]|]}
```

In [20]:
```
let test_produit = automate_produit debut_ab fin_ba;;
```

Out[20]:
```
val test_produit : afd =
  {taille = 9; initial = 0; finals = [8];
   transition =
    [|[(A, 3)]; [(A, 5)]; [(A, 3)]; [(B, 7)]; [(B, 7)]; [(B, 7)];
      [(A, 6); (B, 7); (C, 6)]; [(A, 8); (B, 7); (C, 6)];
      [(A, 6); (B, 7); (C, 6)]|]}
```

In [21]:
```
dot "afd__test_produit.dot" test_produit;;
Sys.command "ls -larth afd__test_produit.dot";;
Sys.command "cat afd__test_produit.dot";;
```

Out[21]:
```
- : unit = ()
```

```
6 → 1, 2-rw-rw-r-- 1 lilian lilian 322 oct.  10 19:05 afd__test_produit.dot
```

Out[21]:
```
- : int = 0
```

```
digraph g {
  node [shape=circle];
  0 → 3 [label=A]
  1 → 5 [label=A]
  2 → 3 [label=A]
  3 → 7 [label=B]
  4 → 7 [label=B]
  5 → 7 [label=B]
  6 → 6 [label=A]
  6 → 7 [label=B]
  6 → 6 [label=C]
  7 → 8 [label=A]
  7 → 7 [label=B]
  7 → 6 [label=C]
  8 → 6 [label=A]
  8 → 7 [label=B]
  8 → 6 [label=C]
}
```

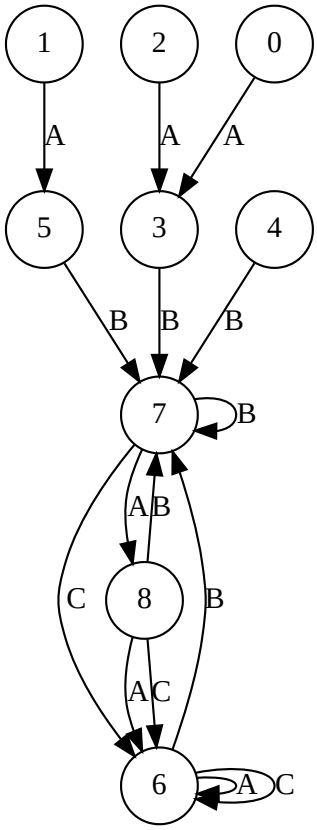
Out[21]:
```
- : int = 0
```

In [22]:
```
Sys.command "dot -Tsvg -o afd__test_produit.svg afd__test_produit.dot";;
Sys.command "ls -larth afd__test_produit.svg";;
```

Out[22]:
```
- : int = 0
```

```
-rw-rw-r-- 1 lilian lilian 8,8K oct.  10 19:05 afd__test_produit.svg
```

Out[22]:
```
- : int = 0
```



On peut vérifier qu'en partant de l'état 0, on doit lire A puis B, et ensuite on lit ce qu'on veut, puis on termine par B puis A.

L'automate produit reconnait l'intersection des deux langages, donc $L(A \times B) = L(A) \cap L(B) = AB\Sigma^* \cap \Sigma^* BA = AB\Sigma^* BA$

Conclusion

Fin. À la séance prochaine. Le TP5 traitera de lambda calcul (en février).