Programmation fonctionnelle

Fiche de TP 8

L3 Informatique 2020-2021

Automates cellulaires

⋆ ⋆ ⋆

Ce TP ressemble dans son déroulement, sa présentation
et le type des questions posées à l'**examen final** à venir.
Il s'agit d'une version révisée et augmentée d'un examen passé.

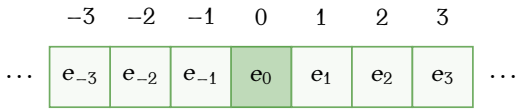
⋆ ⋆ ⋆

1 Explications générales

Nous allons dans ce sujet considérer les automates cellulaires unidimensionnels, des objets informatiques célèbres qui possèdent une très grande richesse. Notre objectif sera d'implanter des fonctions de base pour les manipuler.

Soit A un ensemble non vide quelconque. Un *automate cellulaire unidimensionnel* (abrégé en *automate* dans ce sujet) sur A est la donnée

— d'un *ruban* bi-infini dont les cases contiennent des valeurs de A. Les cases du ruban sont indexées par l'ensemble des entiers. Le ruban est représenté par



— d'une valeur spéciale v de A, appelée *vide* ;

— d'une *fonction évolution* $f : A \times A \times A \to A$. Cette fonction est paramétrée par trois éléments de A et renvoie un nouvel élément de A.

Le principe de fonctionnement de l'automate est le suivant. Son ruban r est rempli avec une configuration initiale (au choix) d'éléments de A. Ensuite, le ruban *évolue* en remplaçant chaque valeur de ses cases par la valeur calculée par la fonction d'évolution f en tenant compte

de ses voisines. Plus précisément, le contenu e_i de chaque case d'indice i du ruban devient $f(e_{i-1}, e_i, e_{i+1})$, pour tout $1 \leqslant i \leqslant n-2$. Ceci est résumé schématiquement par

Notons que comme le ruban est bi-infini, il n'y a pas à prévoir de conditions spéciales aux bords. Par ailleurs, la valeur vide v de l'automate intervient lors de sa création : on ne créera que des automates dans lesquels toutes les cases du ruban sont initialisées à v (la configuration initiale de l'automate est placée après sa création).

Prenons maintenant un exemple simple. Considérons l'automate sur $\mathbb{N}$ où 0 est défini comme la valeur vide et la fonction d'évolution est définie par $f(a, b, c) := a + b$. Alors, depuis la configuration initiale

	−1	0	1	2	3	4	5	
$\cdots$	0	1	0	0	0	0	0	$\cdots$

nous pouvons faire évoluer l'automate itérativement. Voici quatre itérations, la lecture se faisant naturellement de haut en bas :

	−1	0	1	2	3	4	5	
$\cdots$	0	1	0	0	0	0	0	$\cdots$
$\cdots$	0	1	1	0	0	0	0	$\cdots$
$\cdots$	0	1	2	1	0	0	0	$\cdots$
$\cdots$	0	1	3	3	1	0	0	$\cdots$
$\cdots$	0	1	4	6	4	1	0	$\cdots$

Les figures 1, 2 et 3 contiennent d'autres exemples d'automates, dont certains sont plus élaborés et font apparaître des motifs remarquables.

2 Recommandations

— Bien lire le sujet avant de commencer à programmer, en particulier la première partie.

— L'élégance dans la programmation sera appréciée, l'inélégance sanctionnée. Cela peut se manifester dans la bonne utilisation des fonctions sur les listes (`map`, `fold_left`, *etc.*), un bon choix pour les identificateurs et la qualité des algorithmes sélectionnés.

— Il est impératif de respecter les signatures spécifiées.

— Toutes les fonctions demandées sont illustrées par de nombreux tests. Il est très fortement recommandé de vérifier que les résultats des tests des fonctions programmées sont conformes à ceux donnés. Les exemples sont également utiles pour bien comprendre le comportement de chaque fonction.

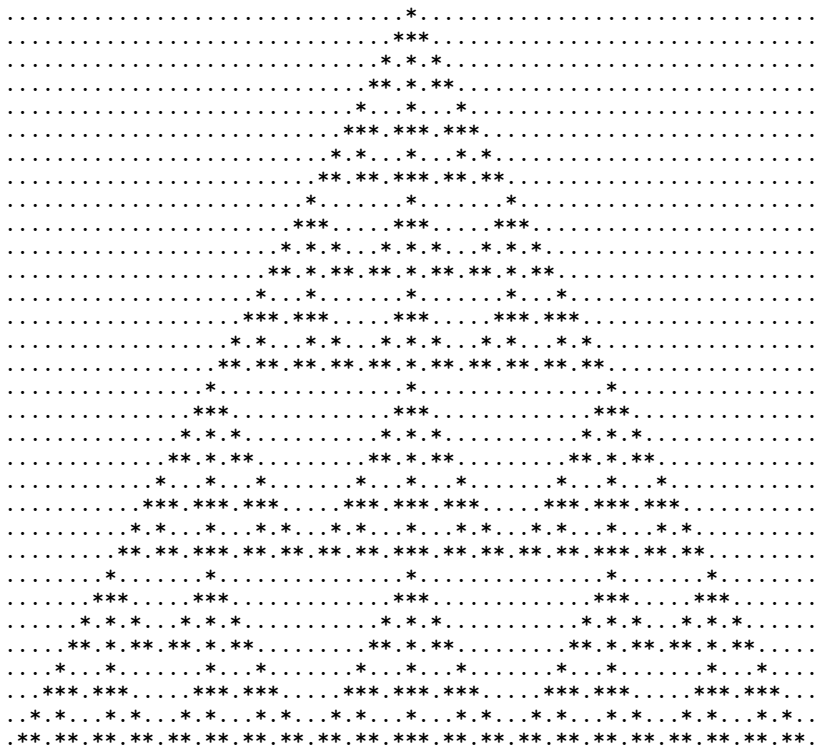


FIGURE 1 – L'automate de « Sierpinski ». Il s'agit d'un automate sur l'ensemble $\{0,1\}$, la valeur vide est 0 et la fonction d'évolution est $f(a,b,c) = (a+b+c) \mod 2$. Les valeurs 0 (resp. 1) sont représentées par des « . » (resp. « * »).

3 Travail à faire

Le travail consiste à compléter le fichier `Automaton.ml` fourni en ajoutant sous chaque commentaire de signature de fonction son implantation correspondante. Les réponses aux éventuelles questions qui ne demandent pas de produire du code doivent être fournies en tant que commentaires dans le fichier. Tout ceci est guidé pas à pas dans les exercices suivants. Le fichier `Automaton.ml` ainsi augmenté est donc l'unique fichier à rendre.

Exercice 1. (Outils)

Nous commençons par programmer quelques fonctions outils, principalement sur les listes, qui nous servirons dans la suite du sujet.

1. Définir une fonction

```
val interval : int -> int -> int list = <fun>
```

qui renvoie la liste d'entiers `[a; a + 1; ...; b]` où `a` et `b` sont ses paramètres.

```
# interval 0 8;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8]

# interval 2 5;;
```

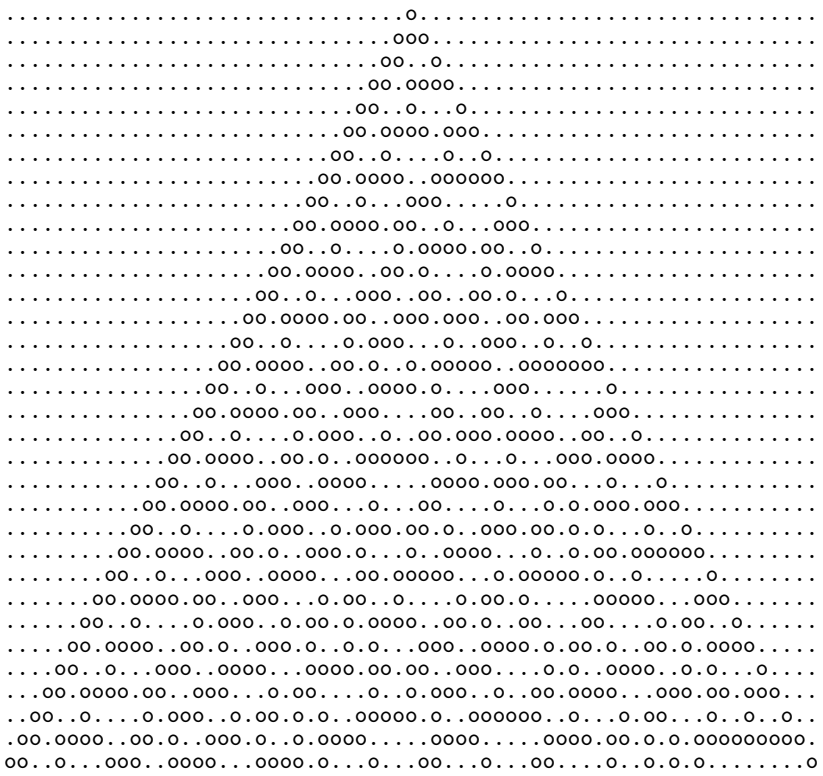


FIGURE 2 – Un automate chaotique. Il s'agit d'un automate sur l'ensemble $\{., o\}$. La valeur vide est « . » et la fonction d'évolution sera décrite plus loin.

```
- : int list = [2; 3; 4; 5]

# interval 3 3;;
- : int list = [3]

# interval (-5) 2;;
- : int list = [-5; -4; -3; -2; -1; 0; 1; 2]

# interval 9 3;;
- : int list = []
```

2. Définir une fonction

```
val string_of_list : 'a list -> ('a -> string) -> string = <fun>
```

qui renvoie une chaîne de caractères représentant les éléments de la liste `lst` en paramètre. Cette fonction est paramétrée de plus par une fonction `f` qui convertit un élément en une chaîne de caractères. La chaîne de caractères est obtenue en concaténant les retours de la fonction `f` appliquée sur chaque élément de `lst`.

```
# string_of_list [0; 1; 2] string_of_int;;
- : string = "012"

```

```
................3...............2.................1..............
...............333.............222...............111.............
..............33333...........22222.............11111............
.............3333333.........2222222...........1111111...........
............333333333.......222222222.........111111111..........
...........33333333333.....22222222222.......11111111111.........
..........3333333333333...2222222222222.....1111111111111........
.........333333333333333.222222222222222...111111111111111.......
........333333333333333332222222222222222.11111111111111111......
.......3333333333333333333222222222222222211111111111111111111.....
......333333333333333333333222222222222222211111111111111111111....
.....33333333333333333333333222222222222222211111111111111111111...
....3333333333333333333333333222222222222222211111111111111111111..
...333333333333333333333333333222222222222222211111111111111111111.
..33333333333333333333333333333222222222222222211111111111111111111
.33333333333333333333333333333332222222222222222111111111111111111
3333333333333333333333333333333332222222222222222111111111111111
33333333333333333333333333333333332222222222222222111111111111111
3333333333333333333333333333333333322222222222222221111111111111
33333333333333333333333333333333333322222222222222221111111111111
3333333333333333333333333333333333333222222222222222211111111111
33333333333333333333333333333333333333222222222222222211111111111
3333333333333333333333333333333333333332222222222222222211111111
33333333333333333333333333333333333333332222222222222222111111111
3333333333333333333333333333333333333333322222222222222222111111
33333333333333333333333333333333333333333322222222222222222111111
3333333333333333333333333333333333333333333222222222222222211111
33333333333333333333333333333333333333333333222222222222222211111
3333333333333333333333333333333333333333333332222222222222222111
33333333333333333333333333333333333333333333332222222222222222111
3333333333333333333333333333333333333333333333322222222222222221
33333333333333333333333333333333333333333333333322222222222222221
3333333333333333333333333333333333333333333333333222222222222222
```

FIGURE 3 – L'automate « max ». Il s'agit d'un automate sur l'ensemble $\mathbb{Z}$, la valeur vide est 0 et la fonction d'évolution est $f(a,b,c) = \max\{a,b,c\}$. Les valeurs vides sont représentées par des « . ».

```
# string_of_list [] string_of_int;;
- : string = ""

# string_of_list [true; false; true] string_of_bool;;
- : string = "truefalsetrue"

# string_of_list [2; 0; 101; 0; 0] (fun x -> if x = 0 then "." else "*");;
- : string = "*.*.."
```

3. Définir une fonction

```
val compose_iter : ('a -> 'a) -> 'a -> int -> 'a list = <fun>
```

paramétrée par une fonction `f`, une valeur `x` et un entier `n` et qui renvoie la liste `[x; f x; f (f x); ...; f (f ...(f x) ...)]`. Il s'agit de la liste de longueur `n + 1` dont le i^e élément est l'image de la composée i^e de `f` appelée sur `x`.

```
# compose_iter (fun x -> x + 1) 0 5;;
- : int list = [0; 1; 2; 3; 4; 5]

# compose_iter (fun x -> -x) 8 6;;
- : int list = [8; -8; 8; -8; 8; -8; 8]

```

```
# compose_iter (fun u -> u ^ "a") "b" 5;;
- : string list = ["b"; "ba"; "baa"; "baaa"; "baaaa"; "baaaaa"]

# compose_iter (fun u -> u) "a" 0;;
- : string list = ["a"]
```

4. Définir une fonction

```
val is_prefix_lists : 'a list -> 'a list -> bool = <fun>
```

qui teste si la liste en 1er paramètre est préfixe de la liste en 2e paramètre.

```
# is_prefix_lists [2; 1; 2; 3] [2; 1; 2; 3; 6; 2; 7];;
- : bool = true

# is_prefix_lists [] [2; 1; 2; 3];;
- : bool = true

# is_prefix_lists [2; 1] [];;
- : bool = false

# is_prefix_lists [] [];;
- : bool = true

# is_prefix_lists ['a'; 'b'; 'b'] ['a'; 'b'; 'b'];;
- : bool = true

# is_prefix_lists [2; 1; 3] [2; 1; 2; 3; 6; 2; 7];;
- : bool = false
```

5. Définir une fonction

```
val is_factor_lists : 'a list -> 'a list -> bool = <fun>
```

qui teste si la liste en 1er paramètre est facteur de la liste en 2e paramètre. On rappelle qu'un facteur d'une liste est une portion contiguë de ses éléments débutant à un endroit arbitraire.

```
# is_factor_lists [2; 1; 3] [4; 2; 1; 3; 4];;
- : bool = true

# is_factor_lists [2; 1; 3] [2; 1; 3; 4];;
- : bool = true

# is_factor_lists [2; 1; 3] [4; 2; 1; 3];;
- : bool = true

# is_factor_lists ['a'; 'a'] ['a'; 'b'; 'a'];;
- : bool = false
```

```
# is_factor_lists [] ['a'];;
- : bool = true

# is_factor_lists [] [];;
- : bool = true
```

6. Définir une fonction

```
val is_subword_lists : 'a list -> 'a list -> bool = <fun>
```

qui teste si la liste en 1er paramètre est sous-mot de la liste en 2e paramètre. On rappelle qu'un sous-mot d'une liste est une liste obtenue en extrayant quelques-uns de ses éléments et en conservant leur ordre d'apparition.

```
# is_subword_lists [1; 3; 5] [1; 2; 3; 4; 5];;
- : bool = true

# is_subword_lists [1; 5; 3] [1; 2; 3; 4; 5];;
- : bool = false

# is_subword_lists ['a'; 'b'] ['a'; 'b'; 'c'];;
- : bool = true

# is_subword_lists [] ['a'];;
- : bool = true

# is_subword_lists [] [];;
- : bool = true
```

7. Définir une fonction

```
val is_duplicate_free : 'a list -> bool = <fun>
```

qui teste si la liste en paramètre contient exactement une occurrence de chacun de ses éléments.

```
# is_duplicate_free [1; 2; 3];;
- : bool = true

# is_duplicate_free [1; 2; 1];;
- : bool = false

# is_duplicate_free [1; 2; 1; 1; 2];;
- : bool = false

# is_duplicate_free [3];;
- : bool = true

# is_duplicate_free [];;
- : bool = true
```

Exercice 2. (**Le type** `automaton`)

Conformément aux explications données plus haut, on définit le type permettant de représenter les automates par le type enregistrement générique

```
type 'a automaton = {
   ribbon : int -> 'a;
   evol : 'a * 'a * 'a -> 'a;
   void : 'a;
}
```

où le champ `ribbon` est le ruban de l'automate représenté comme une fonction qui associe à chaque indice entier `i` la valeur de la case du ruban à l'indice `i`, où le champ `evol` est la fonction d'évolution prenant un triplet de valeurs et renvoyant une valeur et où le champ `void` contient la valeur vide.

1. Définir une fonction

   ```
   val create : ('a * 'a * 'a -> 'a) -> 'a -> 'a automaton = <fun>
   ```

   paramétrée par une fonction d'évolution `evol` et une valeur vide `void`. La fonction renvoie l'automate possédant la fonction d'évolution et valeur vide spécifiée, et dont le ruban contient des valeurs vides en toutes cases.

   ```
   # create (fun (a, b, c) -> a + b + c) 0;;
   - : int automaton = {ribbon = <fun>; evol = <fun>; void = 0}

   # create (fun (a, b, c) -> b) true;;
   - : bool automaton = {ribbon = <fun>; evol = <fun>; void = true}

   # create (fun (a, b, c) -> a ^ b ^ c) "";;
   - : string automaton = {ribbon = <fun>; evol = <fun>; void = ""}
   ```

2. Définir une fonction

   ```
   val get_value : 'a automaton -> int -> 'a = <fun>
   ```

   paramétrée par un automate et un indice. Cette fonction renvoie la valeur du ruban de l'automate en l'indice spécifié.

   ```
   # let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

   # get_value aut1 0;;
   - : int = 0

   # get_value aut1 1024;;
   - : int = 0

   # let aut2 = create (fun (a, b, c) -> b) true;;

   # get_value aut2 (-2048);;
   - : bool = true
   ```

3. Définir une fonction

```
val set_value : 'a automaton -> int -> 'a -> 'a automaton = <fun>
```

paramétrée par un automate, un indice et une valeur. Cette fonction renvoie l'automate obtenu en considérant l'automate en paramètre dans lequel la valeur à l'indice spécifié est la valeur spécifiée.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut2 = set_value aut1 16 4;;

# get_value aut2 15;;
- : int = 0

# get_value aut2 16;;
- : int = 4

# get_value aut1 16;;
- : int = 0
```

Exercice 3. (**Le type** `bunch`)

Le caractère infini des rubans des automates fait qu'il est impossible de demander à les visualiser totalement. Il est cependant possible de visualiser une partie du ruban d'un automate spécifiée par un indice de début et un indice de fin. On appelle cette information une *portion* et est représentée par le type

```
type bunch = int * int
```

où la 1re (resp. 2e) coordonnée contient l'indice de départ (resp. de fin) de la portion.

1. Définir une fonction

```
val get_bunch_values : 'a automaton -> bunch -> 'a list = <fun>
```

paramétrée par un automate et une portion. Cette fonction renvoie, sous forme de liste, l'extrait du ruban de l'automate spécifié la portion.

Indication : utiliser la fonction `interval` de l'exercice 1.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut1 = set_value aut1 3 4;;

# let aut1 = set_value aut1 (-1) 2;;

# get_bunch_values aut1 (-2, 6);;
- : int list = [0; 2; 0; 0; 0; 4; 0; 0; 0]
```

2. Définir une fonction

```
val to_string : 'a automaton -> bunch -> ('a -> string) -> string = <fun>
```

paramétrée par un automate, une portion et une fonction qui convertit une valeur en une chaîne de caractères. Cette fonction renvoie la chaîne de caractère représentant la portion du ruban de l'automate spécifié.

Indication : utiliser la fonction `string_of_list` de l'exercice 1.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut1 = set_value aut1 3 4;;

# let aut1 = set_value aut1 (-1) 2;;

# to_string aut1 (-2, 6) string_of_int;;
- : string = "020004000"

# to_string aut1 (-2, 6) (fun x -> if x = 0 then "." else "*");;
- : string = ".*...*..."
```

3. Définir une fonction

```
val has_factor : 'a automaton -> bunch -> 'a list -> bool = <fun>
```

paramétrée par un automate, une portion et une liste. Cette fonction teste si la portion du ruban de l'automate spécifié contient la liste en tant que facteur.

Indication : utiliser la fonction `is_factor_lists` de l'exercice 1.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut1 = set_value aut1 3 4;;

# let aut1 = set_value aut1 (-1) 2;;

# let aut1 = set_value aut1 5 9;;

# has_factor aut1 (1, 8) [4; 0; 9; 0];;
- : bool = true

# has_factor aut1 (1, 5) [4; 0; 9; 0];;
- : bool = false
```

4. Définir une fonction

```
val has_subword : 'a automaton -> bunch -> 'a list -> bool = <fun>
```

paramétrée par un automate, une portion et une liste. Cette fonction teste si la portion du ruban de l'automate spécifié contient la liste en tant que sous-mot.

Indication : utiliser la fonction `is_subword_lists` de l'exercice 1.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut1 = set_value aut1 3 4;;

# let aut1 = set_value aut1 (-1) 2;;

# let aut1 = set_value aut1 5 9;;

# has_subword aut1 (1, 8) [4; 9];;
- : bool = true

# has_subword aut1 (7, 8) [4; 9];;
- : bool = false
```

Exercice 4. (Transformations et propriétés)

Tout est prêt pour finalement se concentrer sur la fonctionnalité la plus importante des automates, à savoir, leur évolution. Nous allons commencer dans cet exercice par programmer des fonctions de transformation d'automates.

1. Définir une fonction

```
val shift : 'a automaton -> int -> 'a automaton = <fun>
```

paramétrée par un automate et un entier `k`. Cette fonction renvoie l'automate obtenu en décalant le ruban de l'automate spécifié de `k` pas **vers la gauche**.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut1 = set_value aut1 3 4;;

# let aut1 = set_value aut1 (-1) 2;;

# to_string aut1 (-2, 6) string_of_int;;
- : string = "020004000"

# let aut2 = shift aut1 3;;
val aut2 : int automaton = {ribbon = <fun>; evol = <fun>; void = 0}

# to_string aut2 (-2, 6) string_of_int;;
- : string = "004000000"

# let aut3 = shift aut1 (-4);;
val aut3 : int automaton = {ribbon = <fun>; evol = <fun>; void = 0}

```

```
# to_string aut3 (-2, 6) string_of_int;;
- : string = "000002000"

# to_string aut3 (-2, 12) string_of_int;;
- : string = "000002000400000"
```

2. Définir une fonction

```
val mirror : 'a automaton -> 'a automaton = <fun>
```

paramétrée par un automate. Cette fonction renvoie l'automate obtenu en inversant le ruban de l'automate spécifié.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut1 = set_value aut1 3 4;;

# let aut1 = set_value aut1 (-1) 2;;

# to_string aut1 (-8, 8) string_of_int;;
- : string = "00000002000400000"

# let aut2 = mirror aut1;;

# to_string aut2 (-8, 8) string_of_int;;
- : string = "00000400020000000"
```

3. Définir une fonction

```
val map : 'a automaton -> ('a -> 'a) -> 'a automaton = <fun>
```

paramétrée par un automate et une fonction de transformation sur les valeurs. Cette fonction renvoie l'automate obtenu en remplaçant les valeurs des cases du ruban par leurs images par la fonction de transformation.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut1 = set_value aut1 3 4;;

# let aut1 = set_value aut1 (-1) 2;;

# to_string aut1 (-8, 8) string_of_int;;
- : string = "00000002000400000"

# let aut2 = map aut1 (fun x -> x + 1);;

# to_string aut2 (-8, 8) string_of_int;;
- : string = "11111113111511111"
```

4. Définir une fonction

```
val evolution : 'a automaton -> 'a automaton = <fun>
```

paramétrée par un automate et qui renvoie l'automate obtenu en faisant évoluer l'automate spécifié.

```
# let aut1 = create (fun (a, b, c) -> a + b + c) 0;;

# let aut1 = set_value aut1 3 4;;

# let aut1 = set_value aut1 2 1;;

# let aut1 = set_value aut1 (-1) 2;;

# to_string aut1 (-8, 8) string_of_int;;
- : string = "00000002001400000"

# let aut2 = evolution aut1;;

# to_string aut2 (-8, 8) string_of_int;;
- : string = "00000022215540000"
```

5. Définir une fonction

```
val evolutions : 'a automaton -> int -> 'a automaton list = <fun>
```

paramétrée par un automate `aut` et un entier `n`. Cette fonction renvoie la liste des automates obtenus en faisant évoluer itérativement `n` fois l'automate `aut`. Plus précisément, la tête de la liste renvoyé est `aut`, le 2e élément est l'automate obtenu en faisant évoluer `aut`, le 3e est l'automate obtenu en faisant évoluer `aut` deux fois, *etc.*

Indication : utiliser la fonction `compose_iter` de l'exercice 1.

```
# let aut = create (fun (a, b, c) -> a + b) 0;;

# let aut = set_value aut 0 1;;

# let lst = evolutions aut 4;;
val lst : int automaton list =
  [{ribbon = <fun>; evol = <fun>; void = 0};
   {ribbon = <fun>; evol = <fun>; void = 0};
   {ribbon = <fun>; evol = <fun>; void = 0};
   {ribbon = <fun>; evol = <fun>; void = 0};
   {ribbon = <fun>; evol = <fun>; void = 0}]
```

Pour le moment, nous ne visualisons pas dans ce test si les évolutions sont calculées correctement. Ceci nous sera accessible à l'issue de la question suivante.

6. Définir une fonction

```
val evolutions_bunch : 'a automaton -> bunch -> int -> 'a list list = <fun>
```

paramétrée par un automate `aut`, une portion `b` et un entier `n`. Cette fonction renvoie la liste des contenus des rubans sur la portion `b` des automates obtenus en faisant évoluer itérativement `n` fois l'automate `aut`.

```
# let aut = create (fun (a, b, c) -> a + b) 0;;

# let aut = set_value aut 0 1;;

# evolutions_bunch aut (-1, 5) 4;;
- : int list list =
[[0; 1; 0; 0; 0; 0; 0]; [0; 1; 1; 0; 0; 0; 0]; [0; 1; 2; 1; 0; 0; 0];
 [0; 1; 3; 3; 1; 0; 0]; [0; 1; 4; 6; 4; 1; 0]]
```

Cet exemple est celui de l'automate donné dans la 1re partie du sujet. Comme nous pouvons le constater, il permet de calculer le triangle de Pascal.

7. Définir une fonction

```
val is_resurgent : 'a automaton -> bunch -> int -> bool
```

paramétrée par un automate `aut`, une portion `b` et un entier `n`. Cette fonction teste si la portion `b` de `aut` est résurgente au bout d'au plus `n` évolutions. On dit que `b` est *résurgente* en au plus `n` évolutions s'il existe deux automates `aut'` et `aut''` obtenus depuis `aut` par respectivement k et k' évolutions et sont tels que $0 \leqslant k \neq k' \leqslant n$ et les contenus des rubans de `aut'` et `aut''` dans les portions spécifiées par `b` sont identiques.

Indication : utiliser la fonction `is_duplicate_free` de l'exercice 1.

```
# let aut = create (fun (a, b, c) -> a + b) 0;;

# let aut = set_value aut 0 1;;

# is_resurgent aut (-1, -1) 4;;
- : bool = true

# is_resurgent aut (-1, 0) 4;;
- : bool = true

# is_resurgent aut (-1, 1) 4;;
- : bool = false
```

Exercice 5. (Constructions)

Dans cet exercice, nous allons construire quelques exemples d'automates.

1. Définir le nom

```
val sierpinski : int automaton = {ribbon = <fun>; evol = <fun>; void = 0}
```

comme l'automate dont le ruban contient des valeurs entières, la fonction d'évolution vérifie $f(a,b,c) := (a+b+c) \mod 2$ et la valeur vide est 0.

```
# let aut = set_value sierpinski 0 1;;

# print_string (String.concat "\n"
    (List.map
        (fun a -> to_string a (-8, 8) string_of_int)
        (evolutions aut 8)));;
0000000100000000
0000000111000000
0000010101000000
0000110101100000
0001000100010000
0011101110111000
0010100010010100
0110110111010110
1000000010000000- : unit = ()
```

2. Définir le nom

```
val chaos : wb automaton = {ribbon = <fun>; evol = <fun>; void = White}
```

comme l'automate dont le ruban contient les valeurs `White` ou `Black`, la fonction d'évolution vérifie

$$f(a,b,c) = \begin{cases} \text{White} & \text{si } (a,b,c) = (\text{Black},\text{Black},\text{Black}), \\ \text{White} & \text{si } (a,b,c) = (\text{Black},\text{Black},\text{White}), \\ \text{White} & \text{si } (a,b,c) = (\text{Black},\text{White},\text{Black}), \\ \text{Black} & \text{si } (a,b,c) = (\text{Black},\text{White},\text{White}), \\ \text{Black} & \text{si } (a,b,c) = (\text{White},\text{Black},\text{Black}), \\ \text{Black} & \text{si } (a,b,c) = (\text{White},\text{Black},\text{White}), \\ \text{Black} & \text{si } (a,b,c) = (\text{White},\text{White},\text{Black}), \\ \text{White} & \text{si } (a,b,c) = (\text{White},\text{White},\text{White}) \end{cases}$$

et la valeur vide est `White`. Définir au préalable un type somme `wb` qui contient les deux constructeur `White` et `Black`. La fonction d'évolution devra obligatoirement être écrite en mettant en place un filtrage de motifs à bon escient.

```
# let aut = set_value chaos 0 Black;;

# print_string (String.concat "\n"
    (List.map
        (fun a -> to_string a (-8, 8)
            (fun x -> if x = Black then "*" else "."))
        (evolutions aut 8)));;
........*........
```

```
*********.*******
********...******
*******..**.*****
******..*....****
*****..**.***.***
****..*....*...**
***..**.****.**.*
**..*....**......- : unit = ()
```

Exercice 6. (Mémoïsation) — *Bonus (mais bonus très instructif !)*

Fixons pour cet exercice la définition

```
# let aut = set_value sierpinski 0 1;;
```

Ici, `aut` est donc l'automate de Sierpinski (programmé dans l'exercice 5) dans lequel le ruban contient `1` dans sa case d'indice `0` et des `0` partout ailleurs.

1. Essayer de lancer la phrase

   ```
   # evolutions aut 16;;
   ```

   puis la phrase

   ```

   # print_string (String.concat "\n"
     (List.map
        (fun a -> to_string a (-8, 8) string_of_int)
        (evolutions aut 16)));;
   ```

   Résumer ce qu'il se passe (en comparant le comportement obtenu en lançant ces deux phrases) puis proposer une explication rigoureuse.

2. Pour accélérer le calcul du contenu du ruban d'un automate obtenu par plusieurs itérations d'évolution, nous allons recourir à la technique suivante. Pour connaître le contenu d'une case d'indice `i` d'un automate, nous appelons (comme depuis le début du sujet) la fonction `ribbon` de l'automate. Nous allons accompagner cet appel d'une *mise en cache* qui consiste à sauvegarder le résultat dans une liste d'association de sorte que, lors d'un prochain appel, le résultat soit directement donné après recherche dans la liste d'association. On appelle ce procédé *mémoïsation*. Il permet dans la très grande majorité des cas d'obtenir un gain spectaculaire de temps de calcul (au détriment d'un compromis sous forme d'une occupation mémoire souvent plus conséquente).

   Pour mettre en place ce mécanisme, il est fourni la fonction

   ```
   val memo : ('a -> 'b) -> 'a -> 'b = <fun>
   ```

dans le fichier `Memo.ml`. Cette fonction, programmée dans un **paradigme non fonctionnel**, permet de transformer une fonction (non récursive[1]) en sa version mémoïsée. Ceci utilise des mécanismes offert par les fonctions d'ordre supérieur.

Par exemple, considérons la fonction

```
# let f x y =
    (x + y) / 2;;
val f : int -> int -> int = <fun>
```

Pour en obtenir une version mémoïsée, il suffit simplement d'écrire

```
# let f_mem = memo f;;
val f_mem : int -> int -> int = <fun>
```

Cette nouvelle fonction s'utilise exactement comme `f` et se comporte comme cette dernière.

```
# f 1 8, f_mem 1 8;;
- : int * int = (4, 4)

# f 10 20, f_mem 10 20;;
- : int * int = (15, 15)
```

Modifier le code proposé dans les exercices précédents pour faire en sorte de mémoïser quand cela est nécessaire la fonction `ribbon` des automates programmés.

Indication : il y a **très peu** de modifications à faire pour que cela fonctionne. Pour être certain que les modifcations apportées répondent bien à la question, bien tester que le comportement original des automates est respecté et vérifier que la phrase qui demande d'afficher les `16` itérations d'évolution de l'automate de Sierpinski du début de l'exercice s'évalue **instantanément**.

4 Annexes

Le fichier `Automaton.ml` :

```
(*****************************************************************)
(***************************** Exercice 1 ************************)
(*****************************************************************)

(* 1.1. *)
(* val interval : int -> int -> int list = <fun> *)

(* 1.2. *)
```

1. Il existe des variantes de la fonction `memo` adaptées pour la mémoïsation de fonction récursives mais plus complexes à utiliser.

```
(* val string_of_list : 'a list -> ('a -> string) -> string = <fun> *)

(* 1.3. *)
(* val compose_iter : ('a -> 'a) -> 'a -> int -> 'a list = <fun> *)

(* 1.4. *)
(* val is_prefix_lists : 'a list -> 'a list -> bool = <fun> *)

(* 1.5. *)
(* val is_factor_lists : 'a list -> 'a list -> bool = <fun> *)

(* 1.6. *)
(* val is_subword_lists : 'a list -> 'a list -> bool = <fun> *)

(* 1.7. *)
(* val is_duplicate_free : 'a list -> bool = <fun> *)

(*****************************************************************)
(************************** Exercice 2 **************************)
(*****************************************************************)

type 'a automaton = {
    ribbon : int -> 'a;
    evol : 'a * 'a * 'a -> 'a;
    void : 'a
}

(* 2.1. *)
(* val create : ('a * 'a * 'a -> 'a) -> 'a -> 'a automaton = <fun> *)

(* 2.2. *)
(* val get_value : 'a automaton -> int -> 'a = <fun> *)

(* 2.3. *)
(* val set_value : 'a automaton -> int -> 'a -> 'a automaton = <fun> *)

(*****************************************************************)
(************************** Exercice 3 **************************)
(*****************************************************************)

type bunch = int * int

(* 3.1. *)
(* val get_bunch_values : 'a automaton -> bunch -> 'a list = <fun> *)

(* 3.2. *)
(* val to_string : 'a automaton -> bunch -> ('a -> string) -> string
    = <fun> *)

(* 3.3. *)
(* val has_factor : 'a automaton -> bunch -> 'a list -> bool = <fun> *)
```

```
60
61 (* 3.4. *)
62 (* val has_subword : 'a automaton -> bunch -> 'a list -> bool = <fun> *)
63
64 (*****************************************************************)
65 (***************************** Exercice 4 ****************************)
66 (*****************************************************************)
67
68 (* 4.1. *)
69 (* val shift : 'a automaton -> int -> 'a automaton = <fun> *)
70
71 (* 4.2. *)
72 (* val mirror : 'a automaton -> 'a automaton = <fun> *)
73
74 (* 4.3. *)
75 (* val map : 'a automaton -> ('a -> 'a) -> 'a automaton = <fun> *)
76
77 (* 4.4. *)
78 (* val evolution : 'a automaton -> 'a automaton = <fun> *)
79
80 (* 4.5. *)
81 (* val evolutions : 'a automaton -> int -> 'a automaton list = <fun> *)
82
83 (* 4.6. *)
84 (* val evolutions_bunch : 'a automaton -> bunch -> int -> 'a list list
85     = <fun> *)
86
87 (* 4.7. *)
88 (* val is_resurgent : 'a automaton -> bunch -> int -> bool *)
89
90 (*****************************************************************)
91 (***************************** Exercice 5 ****************************)
92 (*****************************************************************)
93
94 (* 5.1. *)
95 (* val sierpinski : int automaton
96     = {ribbon = <fun>; evol = <fun>; void = 0} *)
97
98 (* 5.2. *)
99 (* Type wb. *)
100 (* val chaos : wb automaton
101     = {ribbon = <fun>; evol = <fun>; void = White} *)
102
103 (*****************************************************************)
104 (***************************** Exercice 6 ****************************)
105 (*****************************************************************)
106
107 (* 6.1. *)
108
109 (* 6.2. *)
```

Le fichier `Memo.ml` :

```
let memo f =
    let m = ref [] in
    fun x ->
        try
            List.assoc x !m
        with
            Not_found ->
                let y = f x in
                m := (x, y) :: !m;
                y
```