Programmation fonctionnelle et logique (INF6120)

Devoir 1

Devoir d'entraînement non noté

Samuele Giraudo

Université du Québec à Montréal

Automne 2025

Barème

Donné à titre indicatif uniquement.

- Le TP, dans sa version évaluée, est noté sur 100.
- Chaque question est en moyenne sur 3 points avec des disparités concernant sa difficulté relative, donnée à titre indicatif sur une échelle de ○○○○○ (0/5) à ●●●●● (5/5).
- Toutes les fonctions seront confrontées aux tests donnés dans ce sujet et à d'autres tests supplémentaires similaires. La note attribuée à chaque fonction dépendra de la proportion de passage des tests ainsi que de la qualité de son écriture.
- Environ 20 points sont attribués à la qualité générale du programme produit et au bon maniement du paradigme fonctionnel.

Instructions générales

1. Lire plusieurs fois attentivement le sujet avant de commencer à programmer.
2. 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`, `filter`, *etc.*), les choix judicieux pour les identificateurs et la qualité des algorithmes sélectionnés. Cela peut aussi se manifester par la concision du code, son absence de répétitions et la façon dont les différentes fonctions mises en place sont (ré)utilisées.
3. Il est impératif de respecter les signatures spécifiées.
4. 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 attendu de chaque fonction.
5. Toute utilisation de mécanismes appartenant au paradigme impératif et n'appartenant pas au fonctionnel est interdite sauf mention contraire explicite. Chacune de ces utilisations apporte -25 points. Ceci inclut la mise en place de toute sorte de boucles, de références et de données mutables. Ceci ne concerne pas l'emploi de fonctions d'entrée/sortie qui, elles, sont autorisées.
6. Toute utilisation d'outils génératifs et/ou d'intelligence artificielle est strictement interdite.

Ce document est composé de 20 pages.

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. Nous proposerons une application très simple pour visualiser l'évolution d'un tel automate particulier, l'automate de Sierpinski.

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

	-3	-2	-1	0	1	2	3	
$\cdots$	e_{-3}	e_{-2}	e_{-1}	e_0	e_1	e_2	e_3	$\cdots$

où les valeurs situées en haut des cases sont leurs indices, et chaque e_i est la valeur de la case d'indice i du ruban ;

— d'une valeur fixée 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 $i \in \mathbb{Z}$ du ruban devient $f(e_{i-1}, e_i, e_{i+1})$. Ceci est résumé schématiquement par

$i-1$	i	$i+1$		$i-1$	i	$i+1$
e_{i-1}	e_i	e_{i+1}	$\longrightarrow$	$\cdots$	$f\,(e_{i-1}, e_i, e_{i+1})$	$\cdots$

.

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.

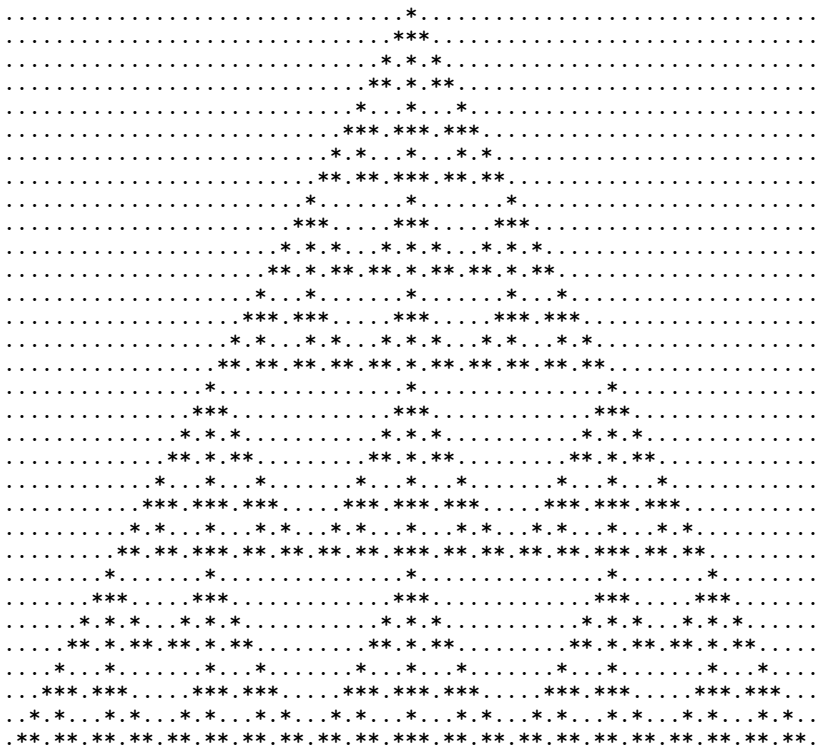

Figure 1 – L'automate de « Sierpinski ». Il s'agit d'un automate sur l'ensemble $\{0, 1\}$, la valeur vide est 0 et la fonction d'évolution est $f(a, b, c) = (a + b + c) \mod 2$. Les valeurs 0 (resp. 1) sont représentées par des « . » (resp. « * »).

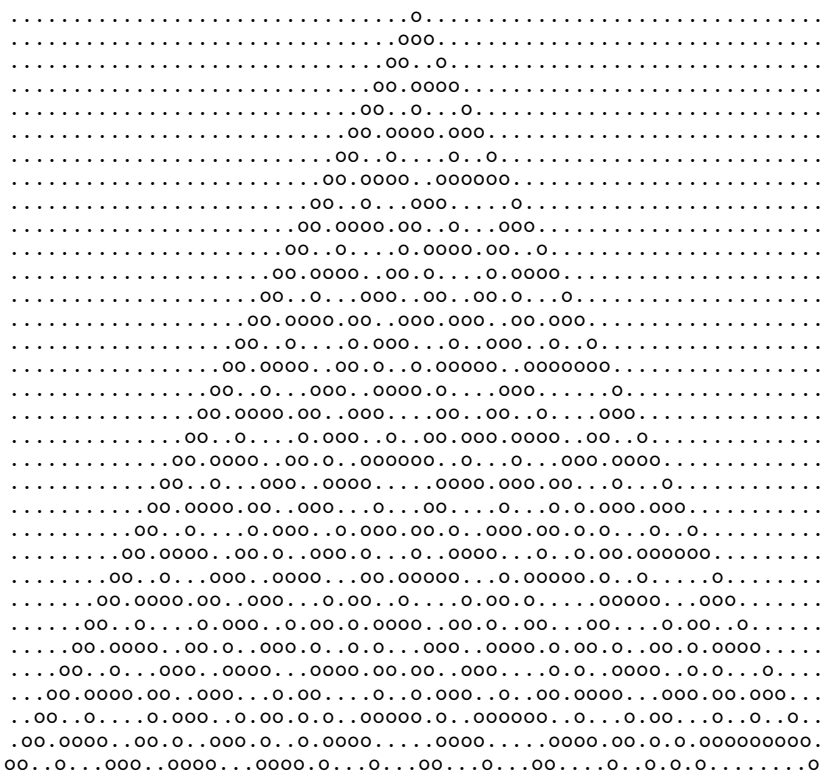

Figure 2 – Un automate chaotique. Il s'agit d'un automate sur l'ensemble {., o}. La valeur vide est « . » et la fonction d'évolution sera décrite plus loin.

2 Travail à accomplir

Le travail consiste à construire un projet `dune` nommé `CellularAutomata` et à le compléter selon les spécifications décrites dans les questions suivantes.

Le projet est à rendre en compressant le répertoire `CellularAutomata` au format `.zip` après l'avoir nettoyé.

Il est important de mentionner, dans le fichier de configuration du projet `dune-project`, votre prénom, nom et code permanent. Ces informations doivent également apparaître dans chaque fichier `.ml` qui fait partie du projet.

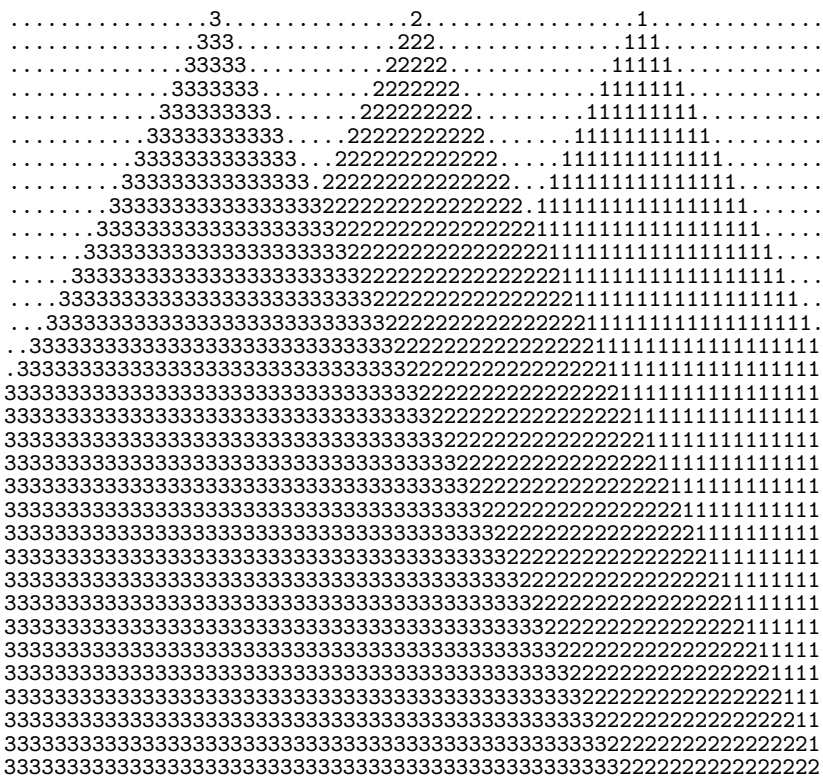

Figure 3 – L'automate « max » (ou encore, l'automate des « pots de peinture renversés »). Il s'agit d'un automate sur l'ensemble $\mathbb{Z}$, la valeur vide est 0 et la fonction d'évolution est $f(a, b, c) = \max\{a, b, c\}$. Les valeurs vides sont représentées par des « . ».

3 Outils

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

Toutes ces fonctions doivent être situées dans un fichier `Tools.ml` placé dans le répertoire `lib` du projet.

Question 3.1. *(●○○○○)* — 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;;
- : 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 = []
```

Question 3.2. *(●○○○○)* — Définir une fonction

```
val list_to_string : ('a -> string) -> 'a list -> 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`.

```
# list_to_string string_of_int [0; 1; 2];;
- : string = "012"
# list_to_string string_of_int [];;
- : string = ""
# list_to_string string_of_bool [true; false; true];;
- : string = "truefalsetrue"
# list_to_string (fun x -> if x = 0 then "." else "*") [2; 0; 101; 0; 0];;
- : string = "*.*.."
```

Question 3.3. *(●○○○○)* — 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"]
```

Question 3.4. *(●○○○○)* — Définir une fonction

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

qui teste si la liste en 1$^{\text{er}}$ paramètre est préfixe de la liste en 2$^{\text{e}}$ 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
```

Question 3.5. *(●○○○○)* — Définir une fonction

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

qui teste si la liste en 1$^{\text{er}}$ paramètre est facteur de la liste en 2$^{\text{e}}$ 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
```

Question 3.6. *(●○○○○)* — 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. Nous rappellons 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
```

Question 3.7. *(●●○○○)* — 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
```

4 Représentation des automates

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

```
type 'a automata = {
    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.

Ce type, ainsi que les fonctions demandées dans cette partie, doivent être situées dans un fichier `Automata.ml` placé dans le répertoire `lib` du projet.

Question 4.1. *(●●○○○)* — Définir une fonction

```
val create : ('a * 'a * 'a -> 'a) -> 'a -> 'a automata = <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 la valeur vide en toutes cases.

```
# create (fun (a, b, c) -> a + b + c) 0;;
- : int automata = {ribbon = <fun>; evol = <fun>; void = 0}
# create (fun (a, b, c) -> b) true;;
- : bool automata = {ribbon = <fun>; evol = <fun>; void = true}
# create (fun (a, b, c) -> a ^ b ^ c) "";;
- : string automata = {ribbon = <fun>; evol = <fun>; void = ""}
```

Question 4.2. *(●○○○○)* — Définir une fonction

```
val get_value : 'a automata -> 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
```

Question 4.3. *(●●○○○)* — Définir une fonction

```
val set_value : 'a automata -> int -> 'a -> 'a automata = <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
```

5 Représentation de portions de rubans

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 bunches = int * int
```

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

Ce type, ainsi que les fonctions demandées dans cette partie, doivent être situées dans un fichier `Bunches.ml` placé dans le répertoire `lib` du projet.

Question 5.1. *(●●○○○)* — Définir une fonction

```
val get_bunch_values : 'a automata -> bunches -> '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.

```
# 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]
```

Question 5.2. *(●●○○○)* — Définir une fonction

```
val to_string : 'a automata -> bunches -> ('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.

```
# 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 = ".*...*..."
```

Question 5.3. *(●○○○○)* — Définir une fonction

```
val has_factor : 'a automata -> bunches -> '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.

```
# 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
```

Question 5.4. *(●○○○○)* — Définir une fonction

```
val has_subword : 'a automata -> bunches -> '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.

```
# 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
```

6 Transformations et propriétés

Tout est finalement prêt pour que nous puissions nous 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.

Ces nouvelles fonctions sont à implanter dans le fichier `Automata.ml` ou dans un nouveau fichier `Evolutions.ml` placé dans le répertoire `lib` du projet selon sera qui sera demandé.

Question 6.1. *(●○○○○)* — Définir dans `Automata.ml` une fonction

```
val shift : 'a automata -> int -> 'a automata = <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 automata = {ribbon = <fun>; evol = <fun>; void = 0}
# to_string aut2 (-2, 6) string_of_int;;
- : string = "004000000"
# let aut3 = shift aut1 (-4);;
val aut3 : int automata = {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"
```

Question 6.2. *(●○○○○)* — Définir dans `Automata.ml` une fonction

```
val mirror : 'a automata -> 'a automata = <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 = "00004000020000000"
```

Question 6.3. *(●○○○○)* — Définir dans `Automata.ml` une fonction

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

paramétrée par une fonction de transformation sur les valeurs et un automate. 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 (fun x -> x + 1) aut1;;
# to_string aut2 (-8, 8) string_of_int;;
- : string = "11111113111511111"
```

Question 6.4. *(●●○○○)* — Définir dans `Evolutions.ml` une fonction

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

paramétrée par un automate et qui renvoie l'automate obtenu en faisant évoluer l'automate spécifié. Ne pas hésiter à introduire de nouvelles fonctions, en particulier ici, une fonction `evol` dans `Automata.ml` permettant d'accéder au champ `evol` d'une valeur de type `automata`.

```
# 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"
```

Question 6.5. *(●●○○○)* — Définir dans `Evolutions.ml` une fonction

```
val evolutions : 'a automata -> int -> 'a automata 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 2^e élément est l'automate obtenu en faisant évoluer `aut`, le 3^e est l'automate obtenu en faisant évoluer `aut` deux fois, *etc.*

```
# 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 automata 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 dès la question suivante.

Question 6.6. *(●●○○○)* — Définir dans `Evolutions.ml` une fonction

```
val evolutions_bunch : 'a automata -> bunches -> 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; 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.

Question 6.7. *(●●○○○)* — Définir dans `Evolutions.ml` une fonction

```
val string_representation : 'a automata -> bunches -> int -> ('a -> string) -> string
```

paramétrée par un automate `aut`, une portion `b`, un entier `n` et une fonction `val_to_string`. Cette fonction renvoie la chaîne de caractères contenant ligne par ligne la portion du ruban de `aut` spécifiée par `b` sur `n` évolutions. La fonction `val_to_string` permet de convertir en chaînes de caractères les valeurs des éléments du ruban.

```
# let aut = create (fun (a, b, c) -> a + b) 0;;
# let aut = set_value aut 0 1;;
# string_representation aut (-1, 5) 4 string_of_int |> print_endline;;
0100000
0110000
0121000
0133100
0146410
- : unit = ()
```

Question 6.8. *(●●●○○)* — Définir dans `Evolutions.ml` une fonction

```
val is_resurgent : 'a automata -> bunches -> 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.

```
# 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
```

7 Constructions

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

Les fonctions demandées ici sont à placer dans un fichier `Examples.ml` situé dans le répertoire `lib` du projet.

Question 7.1. *(●●●○○)* — Définir le nom

```
val sierpinski : int automata = {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;;
# string_representation aut (-8, 8) 8 string_of_int |> print_endline;;
00000000100000000
00000001110000000
00000010101000000
00000110101100000
00001000100010000
00011101110111000
00101000100010100
01101101110110110
10000000100000001
- : unit = ()
```

Question 7.2. *(●●●○○)* — Définir le nom

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

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

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

et la valeur vide est `White`. Définir au préalable un type somme `wb` qui contient les deux constructeurs `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;;
# let val_to_string = (fun x -> if x = Black then "*" else ".");;
# string_representation aut (-8, 8) 8 val_to_string |> print_endline;;
........*........
.......***.......
......**..*......
.....**.****.....
....**..*...*....
...**.****.***...
..**..*....*..*..
.**.****..******.
**..*...***.....*
- : unit = ()
```

8 Mémoïsation

Il s'agit ici de créer un fichier `Memoization.ml` qui va contenir les fonctions vues ici et les réponses en langue naturelle (sous la forme de commentaires en français aux questions demandées).

Fixons pour cet exercice la définition

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

Ici, `aut` est donc l'automate de Sierpinski (programmé dans la partie 7) dans lequel le ruban contient `1` dans sa case d'indice `0` et des `0` partout ailleurs.

Question 8.1. *(○○○○○)* — Essayer de lancer la phrase

```
# evolutions aut 14;;
```

puis la phrase (en restant un peu patient, suivant la puissance de votre ordinateur)

```
# string_representation aut (-8, 8) 14 string_of_int |> print_endline;;
```

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

Pour optimiser 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 en temps de calcul (au détriment d'un compromis sous forme d'une occupation mémoire souvent plus conséquente).

Pour mettre en place ce mécanisme, nous allons utiliser la fonction

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

dont l'implantation est

```
let memo f =
    let memory = ref [] in
    fun x ->
        match !memory |> List.assoc_opt x with
            |None ->
                let y = f x in
                memory := (x, y) :: !memory;
                y
            |Some y -> y
```

Nous allons recopier cette fonction telle quelle dans `Memoization.ml` [a]. Celle-ci, programmée dans un **paradigme non fonctionnel**, permet de transformer une fonction (non récursive [b]) en sa version mémoïsée. Ceci utilise des mécanismes offert par les fonctions d'ordre supérieur.

Par exemple, considérons la fonction

a. Bien évidement, en guise d'exception à ce qui est mentionné dans les instructions générales en 1re page de ce sujet, les éléments impératifs de cette fonction n'apportent aucun malus.

b. Il existe des variantes de la fonction `memo` adaptées pour la mémoïsation de fonction récursives mais plus complexes à utiliser. Elles ne sont pas nécessaires dans le cadre de ce TP.

```
# 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 `f_mem` 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)
```

Question 8.2. *(●●●○○)* — Modifier la fonction `evolution` du fichier `Evolutions.ml` de sorte à mémoïser l'appel à la fonction `ribbon` de l'automate qu'elle considère. Pour tester si votre tentative est correcte, reprendre l'exemple précédent (au tout début de la question 8.1) et s'assurer que le temps de réponse est quasi **instantané**.

9 Fonction principale et rendu final

Nous allons maintenant remplir le fichier `main.ml` situé dans le répertoire `bin` du projet. Le site

https://ocaml.org/docs/cli-arguments

peut être consulté pour des renseignements sur la gestion des arguments.

Question 9.1. *(○○○○○)* — Remplir la fonction principale de `main.ml` de sorte que le programme accepte trois arguments entiers. Les deux premiers spécifient une portion `b` et le dernier un nombre `n` d'itérations. Dans cette fonction, il doit être créé l'automate de Sierpinski dont la case d'indice `0` est initialisée à `1`. Ensuite, les rubans successifs de `n` évolutions de l'automate doivent être affichés, où une valeur de case de `0` sera imprimée par un `.` et une valeur de case de `1` sera imprimée par une `*`.

```
>> dune exec CellularAutomata -- -16 16 15
................*................
...............***...............
..............*.*.*..............
.............**.*.**.............
............*...*...*............
...........***.***.***...........
..........*.*...*...*.*..........
.........**.**.***.**.**.........
........*.......*.......*........
.......***.....***.....***.......
......*.*.*...*.*.*...*.*.*......
.....**.*.**.**.*.**.**.*.**.....
....*...*.......*.......*...*....
...***.***.....***.....***.***...
..*.*...*.*...*.*.*...*.*...*.*..
.**.**.**.**.**.*.**.**.**.**.**.
```

Notons le `--` permettant de séparer l'appel à `dune` des arguments du programme.

Le rendu final est formé des fichiers ainsi créés tout au long du TP, organisés dans une architecture de projet `dune`. Pour résumer, en supprimant les fichiers régénérables, l'arborescence doit être de la forme

```
|- CellularAutomata
    |- bin
    |   |- dune
    |   |- main.ml
    |- CellularAutomata.opam
    |- dune-project
    |- lib
    |   |- Automata.ml
    |   |- Bunches.ml
    |   |- dune
    |   |- Evolutions.ml
    |   |- Examples.ml
    |   |- Memoization.ml
    |   |- Tools.ml
    |- test
        |- dune
        |- test_CellularAutomata.ml
```