Programmation fonctionnelle et logique (INF6120)

Devoir 2

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 et du paradigme logique.

Instructions générales

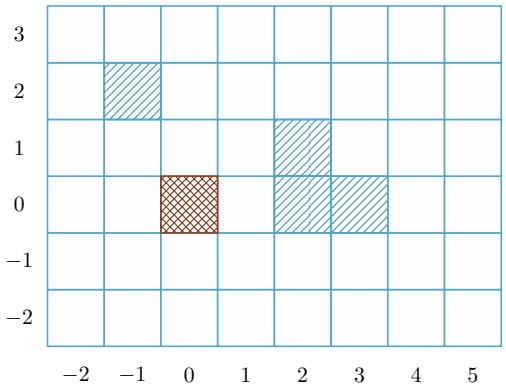
1. Lire plusieurs fois attentivement le sujet avant de commencer à programmer.
2. L'élégance dans la programmation (fonctionnelle et logique) sera appréciée, l'inélégance sanctionnée. Cela peut se manifester dans la bonne utilisation des fonctions sur les listes (`map`, `fold_left`, `filter`, *etc.*), 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 fonctions et prédicats mis en place sont (ré)utilisés.
3. La plupart des fonctions et prédicats demandés sont illustrés par des tests. Il est très fortement recommandé de vérifier que les résultats des tests des fonctions et prédicats programmés sont conformes à ceux donnés. Ceci forme aussi des exemples utiles pour bien comprendre le comportement attendu de chaque fonction et prédicat.
4. Toute utilisation de mécanismes appartenant au paradigme impératif et n'appartenant pas au fonctionnel ou logique 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 ou prédicats d'entrée/sortie qui, eux, sont bien autorisés.
5. Toute aide externe et/ou utilisation d'outils génératifs et/ou d'intelligence artificielle est strictement interdite. Le projet est à faire et à rendre individuellement.

1 Explications générales

L'objectif de ce devoir est, dans une 1re partie, d'implanter en OCaml un générateur aléatoire de polyominos. Le programme conçu va produire un fichier Prolog qui contiendra le codage du polyomino ainsi généré. Dans une 2e partie, notre objectif est d'implanter en Prolog des prédicats permettant de poser des questions de nature combinatoire au polyomino généré.

Commençons ici par poser les définitions de base sur les objets mis en jeu.

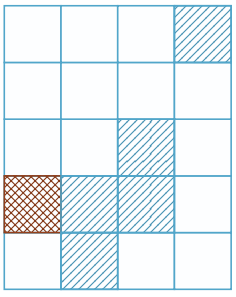
Le *plan cartésien discret* est l'ensemble $\mathbb{Z} \times \mathbb{Z}$. Ses éléments sont des *cases*. Si (x, y) est une case, x est son *abscisse* et y est son *ordonnée*. Une case c est *voisine* d'une case c' si c' est immédiatement en haut, en bas, à gauche ou à droite de c. Le plan cartésien discret est dessiné de sorte à progresser de manière croissante de gauche à droite sur les abscisses et de bas en haut sur les ordonnées. Par exemple,



(1.1)

représente une parcelle (ici, finie) du plan cartésien (infini). L'origine $(0, 0)$ est toujours repérée par une texture faite d'hachures croisées. Quatre cases sont de plus marquées par des hachures diagonales. Celle la plus à droite est la case $(3, 0)$. Dans la suite, nous ne mentionnerons plus les graduations en abscisse et en ordonnée. Elles peuvent se déduire sans ambigüité de la position de l'origine qui elle sera toujours mentionnée avec la même convention.

Un *polyomino* P est un sous-ensemble fini du plan cartésien. Un case (x, y) est *remplie* dans P si $(x, y) \in P$. Par exemple, l'ensemble de cases $P := \{(0, 0), (1, 0), (1, -1), (2, 0), (2, 1), (3, 3)\}$ est un polyomino. Sa représentation graphique est

$P =$  . (1.2)

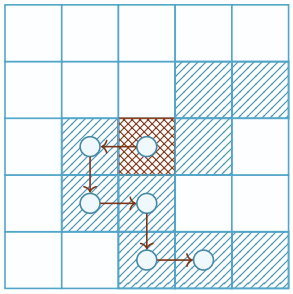
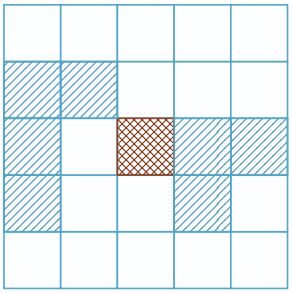
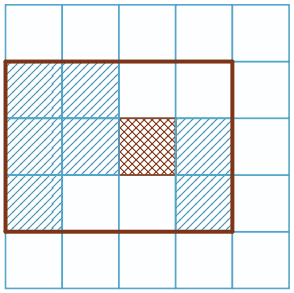
Les cases remplies du polyomino sont marquées par des des hachures diagonales ou par des hachures croisées.

Donnons à présent quelques définitions de base sur ces objets combinatoires[a]. Elle vont nous servir dans la suite, notamment lors des spécifications des fonctions et prédicats à programmer. Soit P un polyomino. L'*aire* de P est son cardinal, ce qui correspond au nombre de ses cases remplies. Le *rectangle englobant* de P est le plus petit rectangle qui contient toutes les cases remplies de P. Un tel rectangle est spécifié par son coin en bas à gauche et son coin en haut à droite. Le polyomino P est *ancré* si l'origine y est remplie.

a. Pour bien intégrer ces définitions, nous conseillons de les synthétiser par des notes personnelles. Cela va être très utile pour bien comprendre les énoncés suivants.

Un *chemin* dans P est une suite $c_1 \dots c_\ell$ de cases remplies de P telle que $\ell \in \mathbb{N}$ et pour tout $1 \leqslant i \leqslant \ell - 1$, la case c_{i+1} est voisine de la case c_i. Ce chemin *connecte* la case c_1 à la case c_ℓ. Par ailleurs, ce chemin est *élémentaire* s'il ne contient aucun doublon. Le polyomino P est *connexe* si pour toutes cases c et c' remplies dans P, il existe un chemin dans P qui connecte c à c'.

Illustrons ces quelques notions par des exemples concrets. Soient les polyominos

$P_1 :=$  , $P_2 :=$  et $P_3 :=$  . (1.3)

L'aire de P_1 est 10, l'aire de P_2 est 8 et l'aire de P_3 est 8. Ces trois polyominos sont ancrés et P_1 et P_3 sont connexes. Un chemin connectant l'origine de P_1 à sa case $(1, -2)$ est dessiné. Le polyomino P_2 n'est pas connexe car, entre autres, sa case $(-2, 0)$ n'est pas atteignable par un chemin partant de son origine. Le rectangle englobant de P_3 est dessiné en lignes épaisses. Il est spécifié par sa case $(-2, -1)$ en bas à gauche et par sa case $(1, 1)$ en haut à droite. Remarquons que cette dernière case n'est pas remplie.

Les polyominos interviennent dans de nombreux contextes en informatique théorique, en combinatoire et en physique statistique. Il est possible de se poser de nombreuses questions sur ces objets. Dans ce devoir, nous allons nous concentrer essentiellement sur trois points :

1. une méthode de génération aléatoire de polyominos ;
2. la détection de motifs (cette notion sera introduite plus loin) ;
3. le calcul de chemins élémentaires.

2 Travail à accomplir

Le travail consiste à construire un projet `dune` nommé `PolyominoGenerator` et à le compléter selon les spécifications décrites dans les questions suivantes. Le projet est à rendre en compressant le répertoire `PolyominoGenerator` au format `.zip` après l'avoir nettoyé par `dune clean` [b]. 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` et `.pl` qui fait partie du projet.

Les exemples mentionnés dans les questions suivantes qui illustrent le comportement de certaines fonctions OCaml seront évalués dans l'interpréteur `utop`. Pour alléger un peu la notation, nous omettrons le préfixe `PolyominoGenerator` dans ces exemples. Ainsi, par exemple, nous écrirons simplement `Square.make` au lieu du plus long mais exact `PolyominoGenerator.Squares.make`. Les exemples qui illustrent le comportement de certains prédicats Prolog seront réalisés dans l'interpréteur `swipl`.

Les questions demandent de développer des fonctions et des prédicats. Ces entités doivent figurer dans le rendu. Il est par ailleurs possible (et souvent conseillé) de définir des types, fonctions ou prédications supplémentaires.

b. Un malus de 10 points est accordé si le répertoire n'est pas nettoyé.

3 Cases

Nous commençons par programmer la représentation des cases. Toutes les entités de cette partie doivent figurer dans un fichier `Squares.ml` placé dans le répertoire `lib` du projet.

Une case est un élément du type imposé

```
type squares = Square of (int * int)
```

Remarquons qu'il s'agit d'un type somme ayant un unique constructeur. Ceci est utile, car comparativement au cas où une case serait représentée par un simple couple d'entiers, les signatures des fonctions vont comporter explicitement le type `squares` au lieu du moins explicite `int * int`. La robustesse du code s'en voit améliorée. En contrepartie, l'accès aux coordonnées d'une case est un peu moins direct.

Question 3.1. *(●○○○○)* — Définir une fonction

```
make : int -> int -> squares
```

telle que `make x y` renvoie une case de coordonnées (`x`, `y`).

Par exemple,

```
utop # Squares.make 2 4;;
- : Squares.squares = Squares.Square (2, 4)
```

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

```
origin : squares
```

telle que `origin` est la case de coordonnées (`0`, `0`).

Par exemple,

```
utop # Squares.origin;;
- : Squares.squares = Squares.Square (0, 0)
```

Question 3.3. *(●○○○○)* — Définir une fonction

```
x_coordinate : squares -> int
```

telle que `x_coordinate sq` renvoie l'abscisse de la case `sq`.

Par exemple,

```
utop # x_coordinate (Squares.make 2 4);;
- : int = 2
```

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

```
y_coordinate : squares -> int
```

telle que `y_coordinate sq` renvoie l'ordonnée de la case `sq`.

Par exemple,

```
utop # y_coordinate (Squares.make 2 4);;
- : int = 4
```

4 Polyominos

Nous programmons maintenant la représentation des polyominos. Toutes les entités de cette partie doivent figurer dans un fichier `Polyominos.ml` placé dans le répertoire `lib` du projet.

Un polyomino est un élément du type imposé

```
type polyominos = Polyomino of (Squares.squares list)
```

Nous adoptons la même convention que celle employée pour représenter les cases consistant à disposer d'un type somme à constructeur unique (voir la partie 3). Ainsi, un polyomino est une constitué d'une liste de cases. Cette liste de cases doit être triée dans l'ordre croissant relativement à l'ordre sur les cases qui consiste à comparer en 1$^{\text{er}}$ lieu les abscisses des cases puis les ordonnées. De plus, la liste ne doit comporter aucun doublon.

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

```
make : squares list -> polyominos
```

telle que `make sq_lst` renvoie le polyomino constitué des cases de la liste `sq_lst`. Il est parfaitement autorisé ici (et ailleurs) d'utiliser les fonctions du module `List`. En particulier, la fonction `List.sort_uniq` est intéressante dans ce contexte.

Par exemple,

```
utop # let sq1 = Squares.make 0 1 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make (-1) 2 in
let sq4 = Squares.make (-2) 8 in
Polyominos.make [sq1; sq2; sq2; sq1; sq4; sq3; sq4];;
- : Polyominos.polyominos = Polyominos.Polyomino
[Squares.Square (-2, 8); Squares.Square (-1, 2); Squares.Square (0, 1); Squares.Square (1, 0)]
```

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

```
origin : Polyominos.polyominos
```

telle que `origin` est le polyomino constitué uniquement de la case origine (`0`, `0`).

Par exemple,

```
utop # Polyominos.origin;;
- : Polyominos.polyominos = Polyominos.Polyomino [Squares.Square (0, 0)]
```

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

```
squares : polyominos -> squares list
```

telle que `squares poly` est la liste des cases du polyomino `poly`.

Par exemple,

```
utop # let sq1 = Squares.make 0 1 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make (-1) 2 in
let sq4 = Squares.make (-2) 8 in
Polyominos.make [sq1; sq2; sq3; sq4] |> Polyominos.squares;;
- : Squares.squares list =
[Squares.Square (-2, 8); Squares.Square (-1, 2); Squares.Square (0, 1); Squares.Square (1, 0)]
```

Question 4.4. *(●●○○○)* — Définir une fonction

```
is_filled_square : squares -> polyominos -> bool
```

telle que `is_filled_square sq poly` teste si le polyomino `poly` possède la case `sq`.

Par exemple,

```
utop # let sq1 = Squares.make 0 1 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make (-1) 2 in
let sq4 = Squares.make (-2) 8 in
let p = Polyominos.make [sq1; sq2; sq3; sq4] in
Polyominos.is_filled_square sq1 p;;
- : bool = true
```

```
utop # let sq1 = Squares.make 0 1 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make (-1) 2 in
let sq4 = Squares.make (-2) 8 in
let p = Polyominos.make [sq1; sq2; sq3; sq4] in
Polyominos.is_filled_square (Squares.make 0 2) p;;
- : bool = false
```

Question 4.5. *(●○○○○)* — Définir une fonction

```
area : polyominos -> int
```

telle que `area poly` est l'aire du polyomino `poly`.

Par exemple,

```
utop # let sq1 = Squares.make 0 1 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make (-1) 2 in
let sq4 = Squares.make (-2) 8 in
Polyominos.make [sq1; sq2; sq3; sq4] |> Polyominos.area;;
- : int = 4
```

Question 4.6. *(●○○○○)* — Définir une fonction

```
add_square : squares -> polyominos -> polyominos
```

telle que `add_square sq poly` est le polyomino dont les cases sont celles du polyomino `poly` avec la case `sq` en plus. Si `poly` possède déjà la case `sq`, la fonction renvoie `poly`.

Par exemple,

```
utop # let sq1 = Squares.make 0 1 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make (-1) 2 in
let sq4 = Squares.make (-2) 8 in
Polyominos.make [sq1; sq2; sq3; sq4] |> Polyominos.add_square (Squares.make 0 3);;
- : Polyominos.polyominos =
Polyominos.Polyomino
[Squares.Square (-2, 8); Squares.Square (-1, 2);
Squares.Square (0, 1); Squares.Square (0, 3); Squares.Square (1, 0)]
```

```
utop # let sq1 = Squares.make 0 1 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make (-1) 2 in
let sq4 = Squares.make (-2) 8 in
Polyominos.make [sq1; sq2; sq3; sq4] |> Polyominos.add_square sq2;;
- : Polyominos.polyominos =
Polyominos.Polyomino
[Squares.Square (-2, 8); Squares.Square (-1, 2);
Squares.Square (0, 1); Squares.Square (1, 0)]
```

Question 4.7. *(●●○○○)* — Définir une fonction

```
bounding_box : polyominos -> (squares * squares) option
```

telle que `bounding_box poly` est une option sur le couple `(sq1, sq2)` de cases où `sq1` est le coin en bas à gauche et `sq2` le coin en haut à droite du rectangle englobant du polyomino `poly`. Dans le cas où `poly` est vide, `None` est renvoyé.

Par exemple,

```
utop # let sq1 = Squares.make 0 1 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make (-1) 2 in
let sq4 = Squares.make (-2) 8 in
Polyominos.make [sq1; sq2; sq3; sq4] |> Polyominos.bounding_box;;
- : (Squares.squares * Squares.squares) option
= Some (Squares.Square (-2, 0), Squares.Square (1, 8))
```

Question 4.8. *(●●●●○)* — Définir une fonction

```
to_string : polyominos -> string
```

telle que `to_string poly` est la chaîne de caractères représentant le polyomino `poly` selon les spécifications suivantes. Si `poly` contient la case origine, elle est dessinée par un `X`. Les autres cases de `poly` sont dessinées par des `O`. Les cases vides sont dessinées par des points médians `·`. La zone affichée correspond au rectangle englobant de `poly`. Comme auparavant, les cases sont dessinées aux endroits spécifiés par leurs coordonnées dans le plan cartésien discret.

Par exemple,

```
utop # let sq1 = Squares.make 0 0 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make 2 0 in
let sq4 = Squares.make (-1) (-1) in
let sq5 = Squares.make 1 1 in
Polyominos.make [sq1; sq2; sq3; sq4; sq5] |> Polyominos.to_string |> print_endline;;
··O·
·XOO
O···
- : unit = ()
```

Pour information, cette fonction est utile pour visualiser les polyominos qui vont être générés dans la suite, mais ne pas fournir cette fonction n'est pas bloquant pour ce qu'il reste à faire.

Question 4.9. *(●●○○○)* — Définir une fonction

```
to_prolog : polyominos -> string
```

telle que `to_prolog poly` est la chaîne de caractères représentant le polyomino `poly` en Prolog selon les spécifications suivantes. Cette chaîne de caractères commence par la représentation en texte fournie par l'appel `to_string poly`, placée entre `/*` et `*/` (pour en faire des commentaires en Prolog). Ensuite, pour chaque case (x, y) de `poly` prises dans l'ordre croissant, la chaîne de caractères est pourvue d'une ligne contenant le fait `filled(square(`x`,`y`)).`.

Par exemple,

```
utop # let sq1 = Squares.make 0 0 in
let sq2 = Squares.make 1 0 in
let sq3 = Squares.make 2 0 in
let sq4 = Squares.make (-1) (-1) in
let sq5 = Squares.make 1 1 in
Polyominos.make [sq1; sq2; sq3; sq4; sq5] |> Polyominos.to_prolog |> print_endline;;
/*
··O·
·XOO
O···
*/

filled(square(-1, -1)).
filled(square(0, 0)).
filled(square(1, 0)).
filled(square(1, 1)).
filled(square(2, 0)).
- : unit = ()
```

Dans le cas où la fonction `to_string` de la question précédente n'a pas été traitée, la 1re partie de l'affichage contenant le commentaire en Prolog du dessin du polyomino pourra être omise sans incidence sur la suite.

5 Génération aléatoire de polyominos

Nous programmons maintenant la génération aléatoire des polyominos. Toutes les entités de cette partie doivent figurer dans un fichier `RandomGenerations.ml` placé dans le répertoire `lib` du projet.

Nous allons utiliser le type imposé

```
type directions =
    |Up
    |Down
    |Left
    |Right
```

permettant de représenter l'une des quatre directions orthogonales. Nous allons aussi utiliser la fonction imposée (qui, notons-le, est un retardateur)

```
let random_direction () =
    match Random.int 4 with
        |0 -> Up
        |1 -> Down
        |2 -> Left
        |_ -> Right
```

dont le but est de générer de manière aléatoire une direction. Ce type et cette fonction doivent être recopiés tels quels dans le projet afin de pouvoir demander un comportement déterministe lors de leurs utilisations. Ceci est important afin d'avoir de pouvoir mettre en place des tests reproductibles et donc exploitables.

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

```
adjacent_square : squares -> directions -> squares
```

telle que `adjacent_square sq dir` est la voisine de la case `sq` selon la direction `dir`.

Par exemple,

```
utop # RandomGenerations.adjacent_square (Squares.make 0 3) RandomGenerations.Up;;
- : Squares.squares = Squares.Square (0, 4)
```

```
utop # RandomGenerations.adjacent_square (Squares.make 0 3) RandomGenerations.Down;;
- : Squares.squares = Squares.Square (0, 2)
```

```
utop # RandomGenerations.adjacent_square (Squares.make 0 3) RandomGenerations.Left;;
- : Squares.squares = Squares.Square (-1, 3)
```

```
utop # RandomGenerations.adjacent_square (Squares.make 0 3) RandomGenerations.Right;;
- : Squares.squares = Squares.Square (1, 3)
```

Question 5.2. *(●●●●●)* — Nous allons maintenant mettre au point une fonction de génération aléatoire de polyominos ancrés et connexes, paramétrée par l'aire $n \geqslant 1$ visée. L'algorithme est très simple et se base sur un processus assez classique étudié en théorie des probabilités, la *marche aléatoire* en dimension 2.

L'algorithme consiste à faire grandir itérativement un polyomino en partant de celui qui ne contient que l'origine. À chaque instant, un case remplie est maintenue marquée. Cette case donne lieu à une nouvelle case marquée choisie aléatoirement de manière uniforme parmi ses quatre voisines. Il se peut que la nouvelle case marquée soit une case remplie. Dans ce cas, la case marquée se déplace une nouvelle fois selon ce même principe. Dès que la case marquée est non remplie, celle-ci est ajoutée au polyomino. Son aire grandit

ainsi de 1 par cette itération. Ce processus est répété jusqu'à ce que le polyomino ait n cases remplies. Par construction, il est facile de voir que le polyomino obtenu ainsi est bien ancré et connexe.

Définir une fonction

```
build_random : int -> polyominos
```

telle que `build_random n` est un polyomino d'aire `n` généré aléatoirement selon l'algorithme qui vient d'être décrit.

Dans les exemples suivants — afin qu'ils soient reproductibles —, nous initialisons avant l'appel à `build_random` la graine du générateur pseudo aléatoire par un appel à la fonction `Random.init`. Si l'algorithme précédent est correctement retranscrit et que la fonction `random_direction` donnée précédemment est bien utilisée, les polyominos générés sont ceux qui apparaissent dans ces tests.

```
utop # Random.init 0;;
- : unit = ()
utop # RandomGenerations.build_random 12 |> Polyominos.to_string |> print_endline;;
OOO·
·OOO
·O·O
OO··
OX··
- : unit = ()
```

```
utop # Random.init 1;;
- : unit = ()
utop # RandomGenerations.build_random 12 |> Polyominos.to_string |> print_endline;;
·OO··
·OO··
·OOOX
OO···
OO···
- : unit = ()
```

```
utop # Random.init 0;;
- : unit = ()
utop # RandomGenerations.build_random 64 |> Polyominos.to_string |> print_endline;;
·············O··
···········OOOO·
···········OOOOO
···········OOOOO
···O···OOOOOOO···
··OO·OOO···OOX···
··O··OO··········
··O··O···········
··OOOO···········
··OOOO···········
·OOOOO···········
·OOOOO···········
OOOOOO···········
·O·OOO···········
- : unit = ()
```

6 Fonction principale du programme OCaml

Pour clôturer cette partie utilisant l'OCaml, nous écrivons la fonction principale du programme. Toutes les entités de cette partie doivent figurer dans un fichier `Main.ml` placé dans le répertoire `bin` du projet.

Question 6.1. *(●○○○○)* — Définir une fonction principale qui permet au programme d'accepter comme argument une aire `n` et un argument facultatif `s` contenant la graine du générateur aléatoire.

Au tout début de l'exécution, dans le cas où `s` est spécifié, la graine du générateur aléatoire doit être positionnée à `s`. Dans le cas contraire, la graine du générateur aléatoire doit être la valeur de l'expression

```
(Unix.time () |> int_of_float) mod 1048576
```

L'accès au module `Unix` est rendu possible en ajoutant « `unix` » à l'élément `libraries` du fichier de configuration `bin/dune` (de sorte ainsi à avoir la ligne `(libraries PolyominoGenerator unix)`).

Ensuite, un polyomino `poly` d'aire `n` doit être généré de manière aléatoire. L'exécution doit créer un fichier `PolyominoGenerator/Prolog/Polyomino.pl` (qui est écrasé s'il existe déjà) et qui doit avoir

```
/*
Area: N
Seed: S
*/

/*
DRAWING
*/

PROLOG
```

comme contenu, où `N` est l'aire `n` de P, `S` est la graine du générateur aléatoire, `DRAWING` est la chaîne de caractères qui représente P et `PROLOG` est le programme Prolog qui représente P. Le contenu de ce fichier doit de plus être imprimé sur la sortie standard. Dans le cas où un argument serait incorrect ou bien qu'il y en ait un nombre non attendu, l'exécution ne doit rien faire d'autre que d'afficher un message d'erreur.

Une documentation complète sur la gestion des arguments est fournie à la page

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

De plus, une documentation sur la création et l'écriture dans les fichiers texte est fournie à la page

https://ocaml.org/docs/file-manipulation

Par exemple, l'exécution suivante, en plus de créer le fichier Prolog comme spécifié, produit l'affichage (tronqué pour garder une taille raisonnable ici) :

```
# dune exec PolyominoGenerator 64 0
/*
Area: 64
Seed: 0
*/

/*
· · · · · · · · · · · · · · O · ·
· · · · · · · · · · · · · · OOOO ·
· · · · · · · · · · · · · · OOOOO
· · · · · · · · · · · · · · OOOOO
· · · O · · · OOOOOOO · · ·
· · OO · OOO · · · OOX · · ·
· · O · · OO · · · · · · · · · ·
· · O · · O · · · · · · · · · · ·
· · OOOO · · · · · · · · · · ·
· · OOOO · · · · · · · · · · ·
· OOOOO · · · · · · · · · · ·
· OOOOO · · · · · · · · · · ·
OOOOOO · · · · · · · · · · ·
· O · OOO · · · · · · · · · · ·
*/

filled(square(-13, -7)).
filled(square(-12, -8)).
...
filled(square(3, 2)).
filled(square(3, 3)).
```

7 Propriétés des polyominos

Toutes les entités de cette partie doivent figurer dans un fichier `Analysis.pl` placé dans le répertoire `Prolog` du projet.

Ce fichier doit inclure la bibliothèque CLP(FD) via

```
:- use_module(library(clpfd)).
```

Ainsi, tous les prédicats définis dans la suite peuvent utiliser des entités apportées par cette bibliothèque de programmation par contraintes.

Le programme `Analysis.pl` doit aussi inclure le fichier `Polyomino.pl` par

```
:- consult("Polyomino.pl").
```

Ainsi, le polyomino étudié est accessible via le prédicat `filled/1` défini dans `Polyomino.pl`. Ce polyomino est nommé *polyomino courant*. Dans la suite, lorsque nous ferons référence à des cases *vides* ou *remplies*, cela correspondra aux cases du polyomino courant ayant cette propriété. Pour les exemples qui vont suivre, nous allons considérer alternativement les deux polyominos courants suivants :

```
/*
Area: 24
Seed: 10
*/

/*
. . . . OOOOO .
XOOOOOOOO .
OO . . . . OO .
. . . . . . . OOO
. . . . . . . OOO
*/

filled(square(0, -1)).
filled(square(0, 0)).
filled(square(1, -1)).
filled(square(1, 0)).
filled(square(2, 0)).
filled(square(3, 0)).
filled(square(4, 0)).
filled(square(4, 1)).
filled(square(5, 0)).
filled(square(5, 1)).
filled(square(6, 0)).
filled(square(6, 1)).
filled(square(7, -3)).
filled(square(7, -2)).
filled(square(7, -1)).
filled(square(7, 0)).
filled(square(7, 1)).
filled(square(8, -3)).
filled(square(8, -2)).
filled(square(8, -1)).
filled(square(8, 0)).
filled(square(8, 1)).
filled(square(9, -3)).
filled(square(9, -2)).
```

```
/*
Area: 24
Seed: 131
*/

/*
OOOO . O . . .
. OOOOOOOO
. . OO . OO . .
. . OOOX . . .
. . . OOO . . .
*/

filled(square(-5, 3)).
filled(square(-4, 2)).
filled(square(-4, 3)).
filled(square(-3, 0)).
filled(square(-3, 1)).
filled(square(-3, 2)).
filled(square(-3, 3)).
filled(square(-2, -1)).
filled(square(-2, 0)).
filled(square(-2, 1)).
filled(square(-2, 2)).
filled(square(-2, 3)).
filled(square(-1, -1)).
filled(square(-1, 0)).
filled(square(-1, 2)).
filled(square(0, -1)).
filled(square(0, 0)).
filled(square(0, 1)).
filled(square(0, 2)).
filled(square(0, 3)).
filled(square(1, 1)).
filled(square(1, 2)).
filled(square(2, 2)).
filled(square(3, 2)).
```

Nous ferons référence à eux dans la suite sous les noms respectifs de *polyomino 10* et de *polyomino 131* (à cause de leur graines respectives de génération aléatoire).

Notons bien, par ailleurs, que les exemples de requêtes suivants sont surtout fournis pour illustrer les spécifications des prédicats demandés. Il est parfaitement possible d'implanter un prédicat de sorte — par exemple — à avoir des réponses dans un ordre différent. Cela est tout à fait correct.

Question 7.1. *(●●○○○)* — Définir un prédicat `add_squares/3` tel que `add_squares(S1, S2, S3)` si l'abscisse de la case `S3` est égale à la somme des abscisses des cases `S1` et `S2` et l'ordonnée de la case `S3` est égale à la somme des ordonnées des cases `S1` et `S2`.

Par exemple,

```
?- add_squares(square(2, 1), square(-4, 5), S).
S = square(-2, 6).
```

```
?- add_squares(square(2, 1), S, square(-4, 5)).
S = square(-6, 4).
```

Question 7.2. *(●●●○○)* — Définir un prédicat `neighbor/2` tel que `neighbor(S1, S2)` si la case `S1` est une voisine de la case `S2`. Ceci ne dépend que de la géométrie du plan cartésien discret. Le polyomino courant n'y a aucune influence.

Par exemple,

```
?- neighbor(S, square(0, 0)).
S = square(0, -1) ;
S = square(0, 1) ;
S = square(-1, 0) ;
S = square(1, 0).
```

```
?- neighbor(square(2, 3), S).
S = square(2, 4) ;
S = square(2, 2) ;
S = square(3, 3) ;
S = square(1, 3).
```

Question 7.3. *(●●○○○)* — Définir un prédicat `filled_neighbor` tel que `filled_neighbor(S1, S2)` si la case `S1` est une voisine de la case `S2` et que la case `S2` est remplie dans le polyomino courant. La case `S1` n'est pas nécessairement remplie.

Par exemple,

```
% Avec le polyomino 10.
?- filled_neighbor(
    square(0, 0), S).
S = square(0, -1) ;
S = square(1, 0) ;
false.
```

```
% Avec le polyomino 10.
?- filled_neighbor(
    square(7, -2), S).
S = square(7, -1) ;
S = square(7, -3) ;
S = square(8, -2) ;
false.
```

```
% Avec le polyomino 131.
?- filled_neighbor(
    square(0, 0), S).
S = square(0, 1) ;
S = square(0, -1) ;
S = square(-1, 0).
```

Question 7.4. *(●●○○○)* — Définir un prédicat `filled_list/1` tel que `filled_list(SQ_LST)` si `SQ_LST` est une liste de cases qui sont toutes remplies dans le polyomino courant.

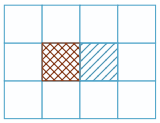
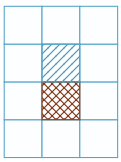
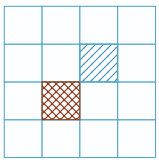
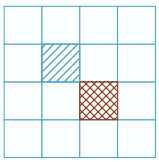
Par exemple,

```
% Avec le polyomino 10.
?- filled_list([]).
true.
```

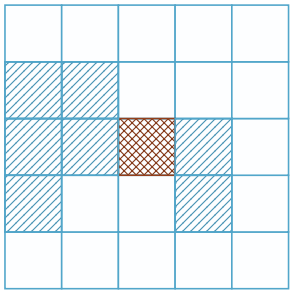
```
% Avec le polyomino 10.
?- filled_list([square(1, 0), square(5, 0), square(7, 1)]).
true.
```

```
% Avec le polyomino 10.
?- filled_list([square(1, 0), square(5, 0), square(-2, 2), square(7, 1)]).
false.
```

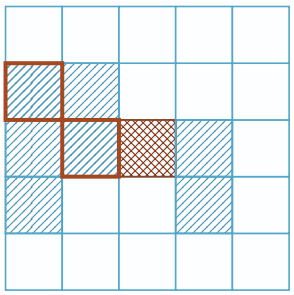
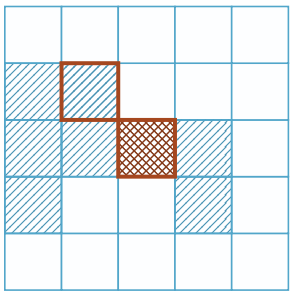
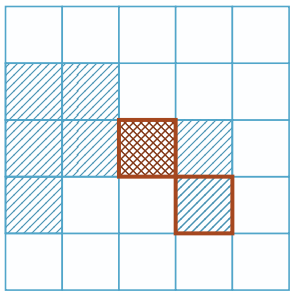
Question 7.5. *(○○○○○)* — Pour énoncer cette question, nous avons besoin de quelques nouvelles définitions. Un *motif* est un polyomino ancré. Un motif peut donc parfaitement être non connexe. Par exemple

$M_1 :=$  , $M_2 :=$  , $M_3 :=$  , $M_4 :=$  (7.1)

sont quatre motifs. Les deux derniers ne sont pas connexes. Un polyomino P possède une *occurrence* d'un motif M s'il est possible de superposer M sur P de sorte que toutes les cases remplies de M sont superposées sur des cases remplies de P. Nous disons de plus que cette occurrence est en *position* (x, y) si, pour ce faire, l'origine de M a été placée sur la case (x, y) de P. Par exemple, soit le polyomino ancré et connexe

$P :=$  . (7.2)

Ce polyomino P est tel qu'il contient trois occurrences du motif M_4, mises en évidence par les cases en gras dans

$P =$  , $P =$  et $P =$  . (7.3)

Remarquons que dans le 1er cas, l'origine de M_4 est confondue avec la case $(-1, 0)$ de P. Ainsi, P admet une occurrence de M_4 en position $(-1, 0)$. De manière similaire, P admet une autre occurrence de M_4 en position $(0, 0)$ et en position $(1, -1)$.

En Prolog, nous allons représenter les motifs par des listes de cases telles que la 1re est l'origine. L'ordre des autres cases n'est pas imposé. Par exemple, le motif M_1 est représenté par la liste

```
[square(0, 0), square(1, 0)]
```

et le motif M_4, par la liste

```
[square(0, 0), square(-1, 1)]
```

Définir un prédicat `pattern_position/2` tel que `pattern_position(PAT, S)` si `PAT` est une liste de cases spécifiant un motif (selon les conventions qui viennent d'être décrites) et `S` est une case telles que le polyomino courant admet une occurrence du motif spécifié en position `S`. Une approche possible pour cela consiste à

1. tout d'abord s'assurer que `S` est bien remplie ;
2. ensuite, construire la liste des case obtenues en ajoutant `S` aux cases de `PAT`, par le biais du prédicat `add_squares/3` développé précédemment. Le prédicat prédéfini `maplist/3` pourra être employé ;
3. finalement, tester si toutes les cases de la liste ainsi obtenue sont remplies, par le biais du prédicat `filled_list/1` développé précédemment.

Par exemple,

```
% Avec le polyomino 10.
?- pattern_position([square(0, 0), square(-1, 1)], S).
S = square(1, -1) ;
S = square(5, 0) ;
S = square(6, 0) ;
S = square(7, -1) ;
S = square(7, 0) ;
S = square(8, -3) ;
S = square(8, -2) ;
S = square(8, -1) ;
S = square(8, 0) ;
S = square(9, -3) ;
S = square(9, -2).
```

```
% Avec le polyomino 10.
?- pattern_position([square(0, 0), square(0, 1), square(0, 2)], S).
S = square(7, -3) ;
S = square(7, -2) ;
S = square(7, -1) ;
S = square(8, -3) ;
S = square(8, -2) ;
S = square(8, -1) ;
false.
```

```
% Avec le polyomino 131.
?- pattern_position([square(0, 0), square(-1, 1)], S).
S = square(-4, 2) ;
S = square(-3, 1) ;
S = square(-3, 2) ;
S = square(-2, -1) ;
S = square(-2, 0) ;
S = square(-2, 1) ;
S = square(-2, 2) ;
S = square(-1, -1) ;
S = square(-1, 0) ;
S = square(-1, 2) ;
S = square(0, -1) ;
S = square(0, 1) ;
S = square(1, 1) ;
S = square(1, 2) ;
false.
```

```
% Avec le polyomino 131.
?- pattern_position([square(0, 0), square(0, 1), square(0, 2)], S).
S = square(-3, 0) ;
S = square(-3, 1) ;
S = square(-2, -1) ;
S = square(-2, 0) ;
S = square(-2, 1) ;
S = square(0, -1) ;
S = square(0, 0) ;
S = square(0, 1) ;
false.
```

Question 7.6. *(●●○○○)* — Définir un prédicat `area/1` tel que `area(A)` si l'entier `A` est l'aire du polyomino courant.

Il est possible (bien que non obligatoire) d'écrire ce prédicat en engendrant toutes les positions du motif `[square(0, 0)]` dans le polyomino courant à l'aide du prédicat prédéfini `findall/3`, puis de compter les solutions par le prédicat prédéfini `length/2`. En effet, compter le nombre d'occurrences de ce motif revient à compter le nombre de cases remplies du polyomino courant.

Par exemple,

```
% Avec le polyomino 10.
?- area(A).
A = 24.
```

```
% Avec le polyomino 10.
?- area(A).
A = 24.
```

Question 7.7. *(●●●●●)* — Définir un prédicat `elementary_path/3` tel que `elementary_path(S_START, S_END, PATH)` si `PATH` est une liste de cases qui commence par la case `S_START`, se termine par la case `S_END` et contient un chemin élémentaire entre ces deux cases du polyomino courant.

Afin d'éviter de s'engouffrer dans des boucles lors de la génération des chemins, il est conseillé de développer un prédicat intermédiaire `elementary_path/4` tel que `elementary_path(S_START, S_END, OPEN, PATH)` si la même spécification que la précédente est vérifiée concernant les arguments `S_START`, `S_END` et `PATH`, avec en plus la condition que toutes les cases de `PATH` figurent dans `OPEN`.

```
% Avec le polyomino 10.
?- elementary_path(square(0, 0), square(1, 0), P).
P = [square(0, 0), square(0, -1), square(1, -1), square(1, 0)] ;
P = [square(0, 0), square(1, 0)] ;
false.
```

```
% Avec le polyomino 10.
?- elementary_path(square(0, -1), square(4, 0), P).
P = [square(0, -1), square(0, 0), square(1, 0), square(2, 0), square(3, 0), square(4, 0)] ;
P = [square(0, -1), square(1, -1), square(1, 0), square(2, 0), square(3, 0), square(4, 0)] ;
false.
```

Question 7.8. *(●●○○○)* — Écrire un prédicat `print_information/0` qui produit comme effet secondaire d'imprimer des informations selon le format

```
Area: A
Domino patterns: DOM_P
Anti-domino patterns: ADOM_P
Diagonal patterns: DIAG_P
Anti-diagonal patterns: ADIAG_P
Paths from the origin: PATHS
```

où

- `A` est l'aire du polyomino courant ;
- `DOM_P` est le nombre d'occurrences du *motif domino*, qui est le motif M_1 défini en (7.1) ;
- `ADOM_P` est le nombre d'occurrences du *motif anti-domino*, qui est le motif M_2 défini en (7.1) ;
- `DIAG_P` est le nombre d'occurrences du *motif diagonal*, qui est le motif M_3 défini en (7.1) ;
- `ADIAG_P` est le nombre d'occurrences du *motif anti-diagonal*, qui est le motif M_4 défini en (7.1) ;
- `PATHS` est le nombre de tous les chemins qui partent de l'origine et qui arrivent à une case arbitraire du polyomino courant.

Pour cela, les prédicats prédéfinits `write/1` et `format/2` pourront être utilisés.

Par exemple,

```
% Avec le polyomino 10.
?- print_information.
Area: 24
Domino patterns: 18
Anti-domino patterns: 14
Diagonal patterns: 11
Anti-diagonal patterns: 11
Paths from the origin: 2619
true.
```

```
% Avec le polyomino 131.
?- print_information.
Area: 24
Domino patterns: 17
Anti-domino patterns: 14
Diagonal patterns: 12
Anti-diagonal patterns: 14
Paths from the origin: 1109
true.
```

8 Application finale

Il s'agit finalement d'achever le projet en fournissant un script BASH pour appeler successivement la génération aléatoire d'un polyomino (par le programme OCaml) et l'impression de ses caractéristiques (par le programme Prolog). Ce script, nommé `Run.sh`, est donné ici :

```
#!/bin/sh

dune exec PolyominoGenerator "$@"
swipl -s Prolog/Analysis.pl -g print_information -g halt
```

Il est à déposer dans le répertoire `PolyominoGenerator`. Ce script prend comme arguments les mêmes que ceux du programme OCaml (voir la question 6.1). Il faut lui attribuer les droits d'exécution par `chmod +x Run.sh`.

Nous obtenons par exemple les affichages suivants :

```
# ./Run.sh 5 0
/*
Area: 5
Seed: 0
*/

/*
·O
OO
OX
*/

filled(square(-1, 0)).
filled(square(-1, 1)).
filled(square(0, 0)).
filled(square(0, 1)).
filled(square(0, 2)).

Area: 5
Domino patterns: 2
Anti-domino patterns: 3
Diagonal patterns: 2
Anti-diagonal patterns: 1
Paths from the origin: 9
```

```
# ./Run.sh 10 1
/*
Area: 10
Seed: 1
*/

/*
·OO··
·OO··
·OOOX
OO···
*/

filled(square(-4, -1)).
filled(square(-3, -1)).
filled(square(-3, 0)).
filled(square(-3, 1)).
filled(square(-3, 2)).
filled(square(-2, 0)).
filled(square(-2, 1)).
filled(square(-2, 2)).
filled(square(-1, 0)).
filled(square(0, 0)).

Area: 10
Domino patterns: 6
Anti-domino patterns: 5
Diagonal patterns: 4
Anti-diagonal patterns: 3
Paths from the origin: 26
```

```
# ./Run.sh 12 17
/*
Area: 12
Seed: 17
*/


/*
OOO··
OO··X
OOOOO
··O··
*/

filled(square(-4, -1)).
filled(square(-4, 0)).
filled(square(-4, 1)).
filled(square(-3, -1)).
filled(square(-3, 0)).
filled(square(-3, 1)).
filled(square(-2, -2)).
filled(square(-2, -1)).
filled(square(-2, 1)).
filled(square(-1, -1)).
filled(square(0, -1)).
filled(square(0, 0)).

Area: 12
Domino patterns: 7
Anti-domino patterns: 6
Diagonal patterns: 5
Anti-diagonal patterns: 4
Paths from the origin: 27
```

9 Rendu final

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

```
|- PolyominoGenerator
    |- Déclaration.md
    |- dune-project
    |- PolyominoGenerator.opam
    |- Run.sh
    |- bin
    |    |- dune
    |    |- Main.ml
    |- lib
    |    |- dune
    |    |- Polyominos.ml
    |    |- RandomGenerations.ml
    |    |- Squares.ml
    |- Prolog
    |    |- Analysis.pl
    |    |- Polyomino.pl
```