Le jeu de la vie

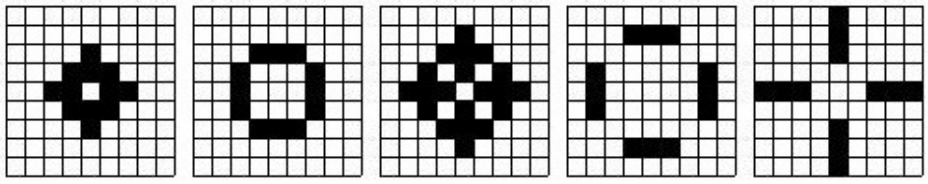
Consignes : À faire à 2 étudiants (au plus) par projet. Pour chaque fonction, **écrire une suite de tests unitaires** si possible (c'est-à-dire, pour les fonctions qui n'ont pas `Unit` comme type de retour), à rendre avec le code du projet. Vous devez écrire en style fonctionnel : pas de boucles, affectations de variables, etc. Chaque enseignant précisera les consignes pour le rendu des projets sur l'ENT.

1 Description

Le jeu de la vie permet de simuler de façon simple l'évolution d'une population au cours du temps. Á partir d'une configuration initiale composée de cellules vivantes (en noir sur les schémas) et mortes (en blanc), on calcule la configuration suivante à partir des règles suivantes.

- Une cellule vivante avec 0 ou 1 cellule voisine vivante meurt à l'itération suivante (sous-peuplement).
- Une cellule vivante avec 2 ou 3 voisines vivantes survit.
- Une cellule vivante avec strictement plus de 3 voisines vivantes meurt (surpeuplement).
- Une cellule morte avec exactement 3 voisines vivantes devient elle-même vivante (naissance).

Le voisinage comprend les diagonales : une cellule a donc 8 voisines. Le schéma ci-dessous donne un exemple d'évolution.



Une manière de programmer le jeu de la vie est de représenter la grille par un tableau à 2 dimensions de booléens, `true` représentant une cellule vivante. On calcule ensuite chaque itération en parcourant le tableau et en appliquant les règles ci-dessus. Cette représentation a deux inconvénients :

- l'évolution du jeu est limité par les dimensions du tableau (que se passe-t-il aux limites ?) ;
- on stocke un tableau potentiellement grand même si seules quelques cellules sont vivantes (perte d'espace mémoire).

Pour palier à ces problèmes, on choisit pour ce TP de représenter une grille par une liste de couples (l, c), chaque couple représentant les coordonnées (ligne, colonne) d'une cellule vivante.

```
type Grille = List[(Int,Int)]
```

2 Entrées/sorties

Avant de programmer le moteur du jeu, il faut pouvoir créer des configurations initiales et afficher des grilles. Pour créer une configuration initiale, on transforme une liste de chaînes de caractères en grille. On suppose que

les chaînes font la même tailles, et sont composées uniquement d'espaces et de "X", qui représentent les cellules vivantes. Par exemple, la liste `l` suivante

```
val l = List(" XX",
             "  X",
             "XXX")
```

doit être transformée en `List((0,1), (0,2), (1,2), (2,0), (2,1), (2,2))` si on numérote les lignes et colonnes à partir de 0.

Question 1 – Écrire la fonction `chainesToGrille(l:List[String]):Grille` qui convertit une liste de chaînes (respectant les contraintes données ci-dessus) en grille.

On souhaite maintenant afficher une grille à l'écran. Pour cela, il faut déterminer les coordonnées du coin supérieur gauche (l_{min}, c_{min}) et du coin inférieur droit (l_{max}, c_{max}) de cette grille. Par exemple, la grille `List((-1,1), (0,1), (1,2), (2,0), (2,1))` est comprise entre les points (-1, 0) et (2, 2). Ensuite, pour chaque point (l, c) dans le rectangle délimité par (l_{min}, c_{min}) et (l_{max}, c_{max}), on affiche un "X" si (l, c) appartient à la grille, une espace sinon. On numérote les lignes de haut en bas, et les colonnes de gauche à droite. Pour la grille `List((-1,1), (0,1), (1,2), (2,0), (2,1))`, on doit donc afficher

```
 X
 X
  X
XX
```

Question 2 – Écrire la fonction `afficherGrille(g:Grille):Unit` qui affiche une grille à l'écran.

3 Moteur de la simulation

On souaite maintenant écrire le moteur qui affiche à l'écran les grilles successives à partir d'une configuration initiale.

Question 3 – Écrire la fonction `voisines8(l:Int, c:Int):List[(Int, Int)]` qui retourne la liste des 8 voisines d'une position (l, c) donnée.

Question 4 – Écrire la fonction `survivantes(g:Grille):Grille` qui prend en paramètre une grille et retourne la liste des cellules qui survivent à l'étape suivante selon les règles données en Section 1.

Question 5 – Écrire la fonction `candidates(g:Grille):Grille` qui prend en paramètre une grille g et retourne la liste des cellules mortes voisines des cellules de g (et qui sont donc susceptibles de naître à l'itération suivante).

Question 6 – En déduire la fonction `naissances(g:Grille):Grille` qui prend en paramètre une grille et retourne la liste des cellules qui naissent à l'étape suivante selon les règles données en Section 1 (attention aux doublons).

Question 7 – Écrire la fonction `jeuDeLaVie(init:Grille, n:Int):Unit` qui prend une configuration initiale (une grille) et un nombre d'étapes n et affiche n itérations de la simulation.

4 Généralisation

D'autres simulations existent construites sur le même principe mais avec des règles et des notions de voisinages différentes. Par exemple, dans *l'automate de Fredkin*, seules quatre cellules sont considérées dans le voisinage (les diagonales ne sont pas prises en comptes), et les règles sont les suivantes :

- une cellule avec un nombre impair de cellules voisines vivantes survit (pour une cellule déjà vivante) ou naît (pour une cellule morte) à l'étape suivante ;
- une cellule vivante avec un nombre pair de voisines meurt à l'étape suivante.

On souhaite donc généraliser les fonctions de la section 3 pour qu'elles acceptent des règles et des voisinages quelconques, comme ceux du jeu de la vie ou de l'automate de Fredkin. Chaque règle peut être vue comme une fonction `Int=>Boolean` qui prend en argument le nombre de voisines vivantes, et renvoie un booléen qui indique si la cellule respectivement naît, survit, ou meurt. Par exemple, la fonction

```
def naitJDLV(nbVoisines:Int):Boolean = (nbVoisines==3)
```

implémente la règle de naissance du jeu de la vie.

Question 8 – Écrire la fonction `voisines4(l:Int, c:Int):List[(Int, Int)]` qui retourne la liste des 4 voisines (nord, sud, est, ouest) d'une position (l, c) donnée.

Question 9 – Écrire les fonctions `naitJDLV` et `survitJDLV` qui correspondent aux règles du jeu de la vie. Faire de même pour les règles de l'automate de Fredkin.

Question 10 – Écrire les fonctions `survivantesG`, `candidatesG` et `naissancesG` (G pour "généralisées") qui correspondent aux fonctions de la section 3, mais qui prennent en paramètres des règles et fonctions de voisinages quelconques.

Question 11 – Écrire la fonction `moteur` qui se comporte comme la fonction `jeuDeLaVie` de la section 3, mais qui prend en paramètres des règles et fonctions de voisinages quelconques.

Question 12 – Utiliser la fonction `moteur` pour (ré)écrire la fonction de simulation du jeu de la vie et de l'automate de Fredkin.

Une variante de l'automate de Fredkin considère comme voisines les 4 cellules nord-est, nord-ouest, sud-est et sud-ouest à la place des cellules nord, sud, est et ouest.

Question 13 – Écrire la fonction de simulation de la variante de l'automate de Fredkin.

5 Bonus

Question 14 – Faire une interface graphique pour l'application avec au minimum un bouton "suivant" pour naviguer entre les configurations (utiliser `scala.swing`).