Listes et Ordre Supérieur

Les bases

Exercice 1 (`iter`)
Écrire la fonction `iter` qui prend en arguments une fonction et une liste et applique cette fonction à tous les éléments de la liste.

```
 val iter : ('a -> unit) -> 'a list -> unit = <fun>

# iter (print_int) [1; 2; 3; 4];;
 1234- : unit = ()
```

Exercice 2 (`map`)
Écrire la fonction `map` qui prend en arguments une fonction et une liste et construit une nouvelle liste avec l'application de la fonction prise en argument sur tous les éléments de la liste.

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

# map (function x -> x * x) [1; 2; 3; 4];;
 - : int list = [1; 4; 9; 16]
```

Exercice 3 (`iteri`)
Écrire la fonction `iteri` qui a le même comportement que la fonction `iter`, sauf que la fonction passée en paramètre a pour premier argument la position de l'élément dans la liste (les listes commencent à 0) et en deuxième argument l'élément lui même.

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

# iteri (function n -> function x -> if n mod 2 = 0 then print_int x) [1;2;3;4;5] ;;
 135- : unit = ()
```

Exercice 4 (`mapi`)
Écrire la fonction `mapi` qui a le même comportement que la fonction `map`, sauf que la fonction passée en paramètre a pour premier argument la position de l'élément dans la liste (les listes commencent à 0) et en deuxième argument l'élément lui même.

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

# mapi (function n -> function x -> power(x,n)) [3;3;3;3;3];;
 - : int list = [1; 3; 9; 27; 81]
```

Exercice 5 (`for_all`)
Écrire la fonction `for_all` qui prend en arguments une fonction booléenne et une liste et renvoie vrai si pour tous les éléments X de la liste, f X = true.

```
 val for_all : ('a -> bool) -> 'a list -> bool = <fun>

# for_all (function x -> x = 0) [0; 0; 0; 0];;
 - : bool = true
```

Exercice 6 (`exists`)
Écrire la fonction `exists` qui prend en arguments une fonction booléenne et une liste et renvoie vrai si pour un des éléments X de la liste, f X = true.

```
 val exists : ('a -> bool) -> 'a list -> bool = <fun>

# exists (function x -> x = 0) [3; 4; 0; 6];;
 - : bool = true
```

Jeu de la Vie

1. Ré-écrire la fonction `draw_cell` du TP précedent, avec en paramètres : la cellule (un entier), ses coordonnées (x,y) sur le plateau ainsi que sa taille et une fonction qui donne la couleur à afficher en fonction de l'état de la cellule. et qui va dessiner correctement la cellule sur le plateau de jeu.
   ```
   val draw_cell : int -> int * int -> int -> (int -> Graphics.color) -> unit = <fun>
   ```

2. Ré-écrire la fonction `draw_board` du TP précédent en utilisant uniquement que les fonctions `iteri` et `draw_cell`. La fonction `draw_board` ne doit pas être elle même récursive.
   ```
   val draw_board : int list list -> int -> (int -> Graphics.color) -> unit = <fun>
   ```

3. Écrire une fonction `remaining` qui determine si il y a au moins une cellule respectant la règle passée en paramètre. La fonction `remaining` ne doit pas être elle même récursive.
   ```
   val remaining : ('a -> bool) -> 'a list list -> bool = <fun>

   # remaining (function x -> x > 0) board ;;
   - : bool = true
   ```

4. Écrire une fonction `map_board` qui va appliquer la fonction passée en paramètre sur chaque case du plateau si celle ci a une valeur supérieure à 0. La fonction `map_board` ne doit pas être elle même récursive.
   ```
   val map_board : 'a list list -> ('a -> 'b) -> 'b list list = <fun>
   ```

5. Écrire une fonction `mapi_board` qui va parcourir chaque case du plateau et qui va appliquer les régles du jeu de la vie dessus. La fonction `mapi_board` ne doit pas être elle même récursive. La fonction prendra en paramètres :
   - le plateau de jeu
   - la règle qui récupère la liste des cellules voisines sous forme de liste
   - la règle qui compte les cellules "actives" dans une liste de cellules
   - la règle qui détermine l'état de la cellule en fonction des cellules "actives"

   ```
   val mapi_board : 'a list list -> (int -> int -> 'a list list -> 'b) ->
      ('b -> 'c) -> ('c -> 'd) -> ('d list list) = <fun>
   ```

6. Écrire une fonction `game_of_life` qui prend en paramètres la largeur du plateau de jeu, la taille d'affichage d'une case, initialise le plateau de jeu correctement, puis tant que le plateau contient des cellules vivantes, affiche ce dernier, applique les différentes règles nécessaires et continue ainsi de suite. La fonction fera donc appel à `mapi_board`.
   ```
   val game_of_life : int -> int -> unit = <fun>
   ```

Bonus

Écrire la fonction `real_life` qui prend en paramètres toutes les fonctions du jeu de la vie, de l'affichage à la fonction qui determine si une cellule reste en vie, afin de rendre le jeu 100% paramètrable.
```
val real_life : '? -> unit = <fun>
```

Modifier les règles du jeu afin que les cellules ne meurent pas directement, mais plutot qu'elles gagnent ou perdent des points de vie en fonction de leurs environnements avoisinant. Il peut être interessant de faire un affichage en fonction de leurs points de vie également.