Jeu de la vie - v1

Introduction du Jeu de la Vie (Game of Life)
`http://en.wikipedia.org/wiki/Conway's_Game_of_Life`
`http://fr.wikipedia.org/wiki/Jeu_de_la_vie`

1 Préliminaires

Les règles

Les cellules seront représentées dans le plateau de jeu par des entiers.

Pour cette première version, les définitions suivantes sont données (pour simplifier les modifications ultérieures) :

```
let cell_color = function
  | 0 -> white          (* nécessite l'ouverture de Graphics *)
  | _ -> black ;;
let new_cell = 1 ;;   (* cellule vivante *)
let empty = 0 ;;
let size_cell = 10 ;;     (* la taille en pixels d'une cellule *)
```

À chaque étape (génération), l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante :

- Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).
- Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt (elle disparaît).

1. Écrire la fonction `is_alive` qui prend une cellule en paramètre et indique si celle-ci est vivante.
   `val is_alive : int -> bool = <fun>`
2. Écrire la fonction `rules` qui à partir d'une cellule et de son nombre de voisines retourne son nouvel état.
   `val rules : int -> int -> int = <fun>`

Listes simples → listes de listes

Le plateau sera représenté par une liste de listes d'entiers (appelée *matrice* ici).

1. Écrire une fonction qui retourne une *matrice* de taille $n \times n$ remplie d'une valeur donnée.

   Exemple d'application :
   ```
   # gen_board 5 0;;
   - : int list list =
   [[0; 0; 0; 0; 0]; [0; 0; 0; 0; 0]; [0; 0; 0; 0; 0]; [0; 0; 0; 0; 0];
    [0; 0; 0; 0; 0]]
   ```
   (On peut commencer par écrire une fonction qui retourne une liste de n valeurs v.)

2. Écrire la fonction `get_cell (x,y) board` qui retourne la valeur en position `(x,y)` dans la matrice `board`. (On peut ici aussi d'abord écrire la fonction qui retourne le $i^{ème}$ élément d'une liste). La fonction retourne la valeur `empty` si l'élément n'existe pas.
   `val get_cell : int * int -> int list list -> int = <fun>`

3. Écrire la fonction `put_cell cell (x,y) board` qui remplace la valeur en `(x,y)` dans la matrice `board` par la valeur `cell`.
   `val put_cell : 'a -> int * int -> 'a list list -> 'a list list = <fun>`

4. Écrire la fonction `count_neighbours (x,y) board` qui retourne le nombre de cellules vivantes (utiliser `is_alive`) autour de la cellule en `(x,y)` dans `board`.
   `val count_neighbours : int * int -> int list list -> int = <fun>`

Fonctions graphiques

Rappels : Tout d'abord, il faut charger le module (à ne faire qu'une seule fois) et ouvrir la fenêtre de sortie :

```
#load "graphics.cma" ;;      (* Load the library *)
open Graphics ;;             (* Open the module *)
open_graph "";;              (* Open the window *)
```

On peut donner en paramètres les dimensions de la fenêtre de sortie (une chaîne de caractères). La fonction suivante permet d'ouvrir une fenêtre de dimensions $size \times size$:

```
let open_window size = open_graph (string_of_int size ^ "x" ^ string_of_int (size+20));;
```

Quelques fonctions utiles (extraits du manuel [1]) :

`val clear_graph : unit -> unit`
Erase the graphics window.

`val rgb : int -> int -> int -> color`
`rgb` r g b returns the integer encoding the color with red component r, green component g, and blue component b. r, g and b are in the range `0..255`.

Exemple : `let grey = rgb 127 127 127 ;;`

`val set_color : color -> unit`
Set the current drawing color.

`val draw_rect : int -> int -> int -> int -> unit`
`draw_rect` x y w h draws the rectangle with lower left corner at x, y, width w and height h. The current point is unchanged. Raise `Invalid_argument` if w or h is negative.

`val fill_rect : int -> int -> int -> int -> unit`
`fill_rect x y w h` fills the rectangle with lower left corner at x,y, width w and height h, with the current color. Raise Invalid_argument if w or h is negative.

Le "plateau" de jeu est une matrice $size \times size$ qui sera affichée sur la fenêtre graphique : il faut faire la correspondance entre les coordonnées dans le plateau et les coordonnées sur la fenêtre graphique.

1. Écrire une fonction qui dessine une cellule (vivante ou morte) à partir de ses coordonnées (sur le plateau de jeu), sa taille (en pixels) et sa couleur : un carré de coté $size$ entouré de gris.
   `val draw_cell : int * int -> int -> Graphics.color -> unit = <fun>`

2. Écrire la fonction `draw_board` qui prend en paramètre la matrice représentant le plateau de jeu, la taille (en pixels) des cellules, et dessine le plateau sur la fenêtre graphique (penser à effacer la fenêtre...).
   `val draw_board : int list list -> int -> unit = <fun>`

1. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html`

2 Le jeu

1. Écrire la fonction `seed_life board size count` qui place aléatoirement (utiliser la fonction `Random.int`) `count` nouvelles cellules dans le plateau `board` de taille `size` × `size`.
   `val seed_life : int list list -> int -> int -> int list list = <fun>`

2. Écrire la fonction `new_board` qui crée un nouveau plateau de jeu à partir de sa taille et du nombre de cellules à placer.
   `val new_board : int -> int -> int list list = <fun>`

3. Écrire la fonction `next_generation` qui à partir du plateau applique les règles du jeu de la vie à toutes les cellules et retourne le nouveau plateau.
   `val next_generation : int list list -> int list list = <fun>`

4. Écrire la fonction `game board n` qui applique les règles du jeu de la vie sur `n` générations au plateau `board` et dessine le plateau à chaque génération.
   `val game : int list list -> int -> unit = <fun>`

5. Écrire enfin la fonction `new_game` qui crée un nouveau jeu à partir de la taille du plateau, du nombre de cellules initiales et du nombre de générations.
   `val new_game : int -> int -> int -> unit = <fun>`

3 Bonus

Quelques ajouts

1. Plutôt que de donner le nombre de générations en paramètres, on peut laisser le jeu tourner tant qu'il reste des cellules vivantes.
   - Écrire la fonction `remaining` qui teste s'il reste des cellules vivantes dans un plateau donné.
   - Modifier la fonction `new_game` : si le nombre de générations passé est 0, le jeu tournera tant qu'il restera des cellules.
2. Il existe des "schémas" connus (le clown, le canon à planeurs). On peut les "charger" à partir d'une liste de coordonnées (voir exemples en ligne).
   - Écrire une fonction `init_pattern pattern size` qui crée un nouveau plateau de jeu de taille `size` à partir de la liste des coordonnées des cellules (`pattern`).
   - Modifier la fonction `new_game` (ou écrire la fonction `new_game_2`) afin qu'elle prenne en paramètre le plateau de jeu, sa taille et le nombre de générations.

Optimisations

1. Réécrire les dernières fonctions en évitant de redessiner la plateau à chaque génération.
2. `count_neighbours` : écrire cette fonction sans utiliser `get_cell` (elle ne doit faire qu'un parcours de la matrice).

Choix et compilation

Utilisez les fonctions d'entrées sorties (`read_int`, `print_...`) pour écrire une version compilée qui laisse le choix entre les différentes versions du jeu.

Voir un exemple en ligne.

Le manuel en ligne risque de vous être utile !