

Jeu de la vie - v1

Introduction au Jeu de la Vie
http://fr.wikipedia.org/wiki/Jeu_de_la_vie

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

```
let new_cell = 1 ;; (* alive cell *)
let empty = 0 ;;
let is_alive cell = cell <> empty ;;
```

1 Boite à outils

Listes simples → listes de listes

Nous travaillons ici avec des listes de listes (appelées *matrices* ou *plateaux*).

1. Écrire la fonction `gen_board` (l , c) *val* qui retourne une *matrice* de taille $l \times c$ remplie de *val*.

```
val gen_board : int * int -> 'a -> 'a list list = <fun>

# gen_board (5, 3) empty ;;
- : int list list = [[0; 0; 0; 0; 0]; [0; 0; 0; 0; 0]; [0; 0; 0; 0; 0]]
```

2. Écrire la fonction `get_cell` (x , y) *board* qui retourne la valeur en position (x , y) dans la matrice *board*.

```
val get_cell : int * int -> 'a list list -> 'a = <fun>
```

3. Écrire la fonction `put_cell` *val* (x , y) *board* qui remplace la valeur en (x , y) dans la matrice *board* par la valeur *val*.

Si la case (x , y) n'existe pas, *board* est retournée inchangée (pas d'exception).

```
val put_cell : 'a -> int * int -> 'a list list -> 'a list list = <fun>
```

4. Écrire la fonction `count_neighbours` (x , y) *board* (l , c) qui retourne le nombre de cellules vivantes (utiliser `is_alive`) autour de la cellule en (x , y) dans *board* de taille (l , c).

```
val count_neighbours : int * int -> int list list -> int * int -> int = <fun>
```

Exemples :

```
# let board = gen_board (5, 3) empty ;;
val board : int list list = [[0; 0; 0]; [0; 0; 0]; [0; 0; 0]; [0; 0; 0]]

# let board = put_cell new_cell (0, 0) board ;;
val board : int list list = [[1; 0; 0]; [0; 0; 0]; [0; 0; 0]; [0; 0; 0]]

# let board = put_cell new_cell (2, 1) board ;;
val board : int list list = [[1; 0; 0]; [0; 0; 0]; [0; 1; 0]; [0; 0; 0]]

# let board2 = gen_board (3, 4) new_cell ;;
val board2 : int list list = [[1; 1; 1; 1]; [1; 1; 1; 1]; [1; 1; 1; 1]]

# count_neighbours (1, 2) board2 (3, 4) ;;
- : int = 8

# count_neighbours (2, 3) board2 (3, 4) ;;
- : int = 3
```

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` : 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 manuel¹) :

- `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.

De la matrice à l'affichage

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 la matrice et les coordonnées sur la fenêtre graphique.

1. Écrire la fonction `draw_cell (x, y) size color` qui dessine une cellule à partir de sa position (*x, y*) dans la matrice, sa taille (en pixels) et sa couleur *color* : un carré de côté *size* entouré de gris. Il est conseillé d'ajouter (1, 1) à (*x, y*) pour ne pas "coller" au cadre.

```
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...).

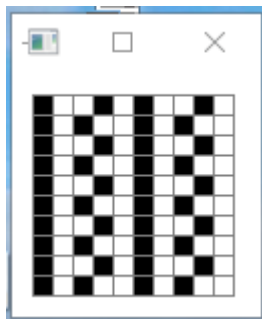
```
val draw_board : int list list -> int -> unit = <fun>
```

Utiliser la définition suivante :

```
let cell_color = function
  | 0 -> white      (* predefined colors in Graphics *)
  | _ -> black ;;
```

1. <https://caml.inria.fr/pub/docs/manual-ocaml-4.05/libref/Graphics.html>

Exemples :



```
# let board = [[1; 1; 1; 1; 1; 1; 1; 1; 1; 1];
               [0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
               [1; 0; 1; 0; 1; 0; 1; 0; 1; 0];
               [0; 1; 0; 1; 0; 1; 0; 1; 0; 1];
               [0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
               [1; 1; 1; 1; 1; 1; 1; 1; 1; 1];
               [0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
               [1; 0; 1; 0; 1; 0; 1; 0; 1; 0];
               [0; 1; 0; 1; 0; 1; 0; 1; 0; 1];
               [0; 0; 0; 0; 0; 0; 0; 0; 0; 0]
               ] ;;

val board : int list list = ...

# let test_display board cell_size =
  open_window (length board * cell_size + 40) ;
  draw_board board cell_size ;;
val test_display : int list list -> int -> unit = <fun>

# test_display board 10 ;;
- : unit = ()
```

2 Le jeu

Les règles

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

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

Écrire la fonction `rules0` qui à partir d'une cellule et de son nombre de voisins retourne son nouvel état.

```
val rules0 : cell:int -> near:int -> int = <fun>
```

La vie

1. Écrire la fonction `seed_life board size nb_cell` qui place aléatoirement (utiliser la fonction `Random.int`) `nb_cell` nouvelles cellules dans le plateau `board` de taille `size × size`.

```
val seed_life : board:int list list -> size:int -> nb_cell: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 : size:int -> nb_cell: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 : board: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 : board:int list list -> n:int -> unit = <fun>
```

Utiliser la définition suivante :

```
let cell_size = 10 ;; (* cell size in pixels *)
```

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 : size:int -> nb_cell:int -> n:int -> unit = <fun>
```

3 Bonus

Quelques ajouts

Infinite life...

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. (Risque de récursion infinie!)

Patterns

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`).
- Écrire la fonction `new_game_pattern` qui lance le jeu avec un "pattern" donné : 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!