# Game of Life - v1

Game of Life introduction
`http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

This first version uses the following definitions (to simplify later modifications):

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

## 1 Toolbox

### Lists → lists of Lists

Here, we work on list of (called *matrices* or *boards*).

1. Write the function `gen_board` $(l,\ c)$ *val* that generates a matrix of size $l \times c$ filled with *val*.

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

2. Write the function `get_cell` $(x,\ y)$ *board* that returns the value at position $(x,\ y)$ in the matrix *board*.

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

3. Write the function `put_cell` *val* $(x,\ y)$ *board* that replaces the value at $(x,\ y)$ in *board* by the value *val*.
If the cell $(x,\ y)$ does not exist, *board* is returned unchanged (no exception).

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

4. Write the function `count_neighbours` $(x,\ y)$ *board* that returns the number of alive cells (use `is_alive`) around the cells at position $(x,\ y)$ in *board* with size $(l,\ c)$.

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

---

*Examples:*

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

## Graphic Functions

**Reminder:** First you will need to load the module (only once) and open the graphics window:

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

`open_graph`: The graphics window size can be given as a string parameter. The following function opens a *size* × *size* window:

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

### Some useful functions (extract from 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`.

  Example: `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.

### From matrix to display

The "board" is a *size* × *size* matrix that will be displayed on the graphics window: it requires to make the correspondence between coordinates in the matrix and those in the graphics window.

1. Write a function that draws a cell (dead or alive) given its coordinates (on the board), its size and its color: a grey square with given *size* filled with *color*.

   It is recommended to add (1,1) to ($x$, $y$) to not "stick" to the frame!

   ```
   val draw_cell : int * int -> int -> Graphics.color -> unit = <fun>
   ```

2. Write the function `draw_board`: it takes as parameters the board (the matrix), the cell size (pixels), and draws the board on the graphics window (don't forget to clear it...).

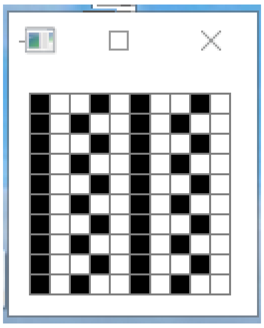
   ```
   val draw_board : int list list -> int -> unit = <fun>
   ```

   Use the following definition:

   ```
   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

*Examples:*



```
# 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 The Game

## Rules

At each step in time (generation), the following transitions occur:

- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.
- Any live cell with two or three live neighbours lives on to the next generation. Otherwise it dies.

Write the function `rules` that takes a cell and its number of neighbours as parameters. It returns the new state of the cell.

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

## Life

1. Write the function `seed_life board size nb_cell` that places `nb_cell` new cells randomly (use the function `Random.int`) in the `size` × `size` matrix `board`.

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

2. Write the function `new_board` that returns a new board from its size and the number of cells.

   ```
   val new_board : size:int -> nb_cell:int -> int list list = <fun>
   ```

3. Write the function `next_generation` that applies the game of life rules to every cell in the board given as parameter. Its returns the new board.

   ```
   val next_generation : board:int list list -> int list list = <fun>
   ```

4. Write the function `game board n` that applies the game of life rules during `n` generations on `board`. It draws the board at each generation.

   ```
   val game : board:int list list -> n:int -> unit = <fun>
   ```

   Use the following definition:

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

5. Finally, write the function `new_game` that creates a new game from the size of the board, the number of cells et the number of generations.

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

# 3 Bonus

## Some add-ons

### Where there's life...

Instead of running the game during a given number of generations, it is possible to let it run as long as alive cells remain.

- Write the function `remaining` that tests whether alive cells remain in the given board.
- Change the function `new_game`: if the generation number given as parameter is 0, the game will continue as long as alive cells remain. (Looping recursion might happens!)

### Patterns

There exist some known "patterns" (the clown, the glider gun). They can be "loaded" from a list of cell coordinates (see examples online).

- Write a function `init_pattern pattern size` that creates a new board with the given `size` from a list of cell coordinates (`pattern`).
- Write the function `new_game_pattern` that runs the game with a given "pattern": it takes the board, its size and the generation number as parameters.

## Optimisations

1. Write again the last functions in order to avoid drawing the whole board at each generation.
2. `count_neighbours`: write this function without using `get_cell` (it must traverse the matrix only once).

## Choices and compilation

Use the input/output functions (`read_int`, `print_...`) to write a compiled version that gives the choice between the different versions of your game.

Have a look at the online example.

The online manual should be useful here!