Game of Life

Reminder : it is strictly FORBIDDEN to use the @ operator.

"The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The "game" is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves or, for advanced players, by creating patterns with particular properties."

– wikipedia

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbours [1]. At each step in time, the following transitions occur :

1. any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. any live cell with two or three live neighbours lives on to the next generation.
3. any live cell with more than three live neighbours dies, as if by overcrowding.
4. any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The goal of this practical is to make your own version of the original game of life in `Caml` using a graphics display. To store our cells we will use the `list` type, each element will be a cell represented by an integer `0` or `1` depending if the cell is Dead or Alive [2]

1. which are the cells that are horizontally, vertically, or diagonally adjacent
2. ©Tecmo

Stage 0 - Generators

0.1 List

Write `gen_list` which generates a list of size `n` containing only `0`s.

```
# gen_list 10 ;;
- : int list = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0]
```

0.2 Random list

Write `gen_rand_list` which generates a list of size `n` containing randomly[3] `0`s and `1`s.

```
# gen_rand_list 10 ;;
- : int list = [0; 1; 0; 1; 0; 0; 0; 0; 1; 1]
```

0.3 List List

Write `gen_board` which generates a "square board" of dimension `n` containing only `0`s.

```
# gen_board 3 ;;
- : int list list = [[0; 0; 0]; [0; 0; 0]; [0; 0; 0]]
```

0.4 Random List List

Write `gen_rand_board` which generates a "square board" of dimension `n` containing randomly `0`s and `1`s.

```
# gen_rand_board 3 ;;
- : int list list = [[0; 1; 0]; [1; 1; 0]; [0; 0; 1]]
```

Stage 1 - Tools

1.1 Count

Write `cell_count` which returns the number of alive cells in the given list.

```
# cell_count [0; 1; 0; 1; 0; 0; 0; 0; 1; 1] ;;
- : int = 4
```

1.2 Re-Count

Write `remaining` which returns the number of alive cells in the given board.

```
# remaining [[0; 1; 0]; [1; 1; 0]; [0; 0; 1]] ;;
- : int = 4
```

3. use `Random.int`

Stage 2 - Display

In this section you will be able to choose the display color[4] of the living and dead cells. You will have to use the functions in this section together and not recode everything each time.

2.1 Square

Write `draw_square (x,y) size` which displays a black square of dimension `size` at the `(x,y)` position.

```
val draw_square : int * int -> int -> unit = <fun>
```

2.2 Square - again

Write `draw_fill_square (x,y) size color` which displays a filled `color` square of dimension `size` at the `(x,y)` position.

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

2.3 Cell

Write `draw_cell (x,y) size cell` which display a cell (dead or alive) of `size` at the `(x,y)` position.

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

2.4 Board

Write `draw_board board size` which display the board with cell's size `size`.

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

Stage 3 - Accessors

For this section, the functions will return an exception (using `failwith`) when it is not possible.

3.1 Cell

Write `get_cell (x,y) board` which returns the value at the cell `(x,y)` of the board if possible.

```
# get_cell (2,1) ([[0;1;2];[3;4;5];[6;7;8]]) ;;
- : int = 3
```

3.2 Replace

Write `replace_cell value (x,y) board` which changes the value of the cell `(x,y)` if possible.

```
# replace_cell 9 (2,1) ([[0;1;2];[3;4;5];[6;7;8]]) ;;
- : int list list = [[0; 1; 2]; [9; 4; 5]; [6; 7; 8]]
```

3.3 Seed life

Write `seed_life board n` which generates `n` new cells on the board.

```
# seed_life [[0; 0; 0; 0]; [0; 0; 0; 0]; [0; 0; 0; 0]; [0; 0; 0; 0]] 3 ;;
- : int list list = [[0; 1; 0; 0]; [0; 0; 1; 0]; [1; 0; 0; 0]; [0; 0; 0; 0]]
```

3.4 Neighborhood

Write `get_cell_neighborhood (x,y) board` which returns the list of the cell's neighbours.

```
# get_cell_neighborhood (2,1) [[11;12;13;14];[21;22;23;24];[31;32;33;34];[41;42;43;44]];;
- : int list = [11; 12; 22; 31; 32]
```

4. or more if you want

Stage 4 - Game

4.1 Iteration

Write `iterate` which returns the board after one transition (one apply of the rules [5])

```
# iterate [[1; 0; 0; 0]; [0; 1; 1; 0]; [0; 0; 0; 0]; [0; 0; 1; 0]];;
- : int list list = [[0; 1; 0; 0]; [0; 1; 0; 0]; [0; 1; 1; 0]; [0; 0; 0; 0]]
```

4.2 Play

Write `play` which displays the board, does a transition and repeats this until there is no more cell alive.

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

Stage 5 - Bonus

5.1 Health Points

We will consider that a new cell start its life with ten health points.

5.1.1 Rules

We will change the previous rules for those :

1. any live cell with fewer than two live neighbours looses two health points
2. any live cell with two live neighbours survives the transition
3. any live cell with three live neighbours wins an health point
4. any live cell with more than three live neighbours looses five health points
5. any dead cell with exactly three live neighours becomes a new live cell with ten health points

5.1.2 Evolving

You will have to re-implement all the functions according to the change to handle cells with health points. You will have also to modify the display to reflect the current number of health points of a living cell.

5.2 Creator

You need to be able to generate new living cell using only your mouse click.

5.3 A new dimension

You need to be able to play on a three dimensional board.

5.4 And many more to come

If you still do not have enough, or you do not like ours, you can also choose to implement your own. Each of those must be well explain and commented in the code.

5. read introduction again.