- TD -
Game of life

The *game of life* models is a simple evolutionary system, and is played on a two-dimensional board. Each square on the board is either empty, or contains a single living cell. For example, the so-called *glider* configuration is as follows:

			O	
	O		O	
		O	O	

Each square on the board has eight immediate neighbours. The board behaves as a torus w.r.t. to the squares on the edge. For example, for a 5 by 5 board the right neighbour of the square at position (5,5) is situated at position (5,0), while the down neighbour is situated at position (0,5).

A generation consists fo the living cells on the board. The next generation is obtained by simultaneously applying the following rules:

- a live cell with fewer than two live neighbours dies, as if by under-population.
- a live cell with two or three live neighbours lives on to the next generation.
- a live cell with more than three live neighbours dies, as if by overpopulation.
- a dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The next generation of the glider configuration is:

		O		
			O	O
		O	O	

This exercise is strongly inspired from the one proposed in Chapter 10 of the book Programming in Haskell, Graham Hutton, Cambridge University Press, 2016.

1. Screen utilities

Control characters can be used to implement useful screen utilities. For example, when displaying the control sequence `\ESC[2J` the screen is cleared and thus, the following action clears the screen:

```
cls :: IO ()
cls = putStr "\ESC[2J"
```

Positioning the cursor at position (x,y), with (1,1) the position at the top-left corner of the screen, is accomplished by displaying the control sequence `\ESC[y;xH`. If we consider `type Pos = (Int, Int)` then, the following action positions the cursor at (x,y):

```
goto :: Pos -> IO ()
goto (x, y) = putStr ("\ESC[" ++ show y ++ ";" ++ show x ++ "H")
```

Define a function `writeat` which writes a string at a given position:

```
writeat :: Pos -> String -> IO ()
```

2. The board

Define a type `Board` which models the board for the game.

Hint: there are relatively few living cells at a given moment and thus, the matrix modelling the board is parse; the implementation using a two-dimensional array wouldn't be efficient in terms of space and an implementation using a list of living cells (and the size of the board) could be considered.

Define a function `showcells` which prints the board:

```
showcells :: Board -> IO ()
```

Define the functions `isAlive` and `isEmpty` which test if there is a, respectively there is no, living cell at a given position:

```
isAlive :: Board -> Pos -> Bool

isEmpty :: Board -> Pos -> Bool
```

Define a function `neighbours` which returns the list of neighbouring positions for a given position (taking into account the width and height of the board):

```
neighbours :: Pos -> [Pos]
```

Hint: you could start by computing naively the list of neighbours as if the position is not on the edge of the board and use then a function which potentially adjusts the positions in the list.

3. The evolution

Define a function `survivors` which leaves on the board only the survivors after one evolution step:

```
survivors :: Board -> Board
```

Hint: we could use another function `livingneighbours :: Board -> Pos -> Int` which computes the living neighbours of the square at a given position.

Define a function `births` which adds the new borns after one evolution step:

```
births :: Board -> Board
```

Hint: for an efficient implementation we should notice that a new born can appear only at the proximity of a living cell.

Define a function `nextgen` which computes the next generation with respect to a board:

```
nextgen :: Board -> Board
```

4. The game

Define a function `life` which plays the game, that is clears the screen, shows the living cells in the current board, waits for a moment, and then continues with the next generation:

```
life :: Board -> IO ()
```

Hint: to wait for a while we can perform some dummy actions using the function

```
wait :: Int -> IO ()
wait n = sequence_ [return ()  | _ <- [1..n*1000] ]
```

Play the game using, for example, the following boards:

```
glider :: Board
glider = [(4,2), (2,3), (4,3), (3,4), (4,4)]
clock :: Board
clock = [(2,1), (3,2), (4,2), (1,3), (2,3), (3,4)]
pentadecathlon :: Board
pentadecathlon=[(1,2),(2,1),(2,2),(2,3),(7,1),(7,2),(7,3),(8,2)]
form :: Board
form=[(1,1),(1,2),(2,2),(2,3),(1,3),(5,1),(5,2),(4,2),(4,1),(5,3)]
```