Functional Programming and Specification
Practical 2

This is an assessed practical exercise, to be completed by **4pm on Friday 27th February**. Please use Moscow ML to test your programs before submitting them, using the DICE `submit` command, like so:

```
submit cs3 fps 2 myfile1 myfile2 ... myfilen
```

For this exercise, you may (but are not required to) work together with *one* other person of your choice. If you do this, you must submit a joint solution bearing both names rather than two separate copies.

⋆

You have probably heard of Conway's Game of Life. The game — really a simulation of population growth rather than a game in the usual sense — is conducted on a rectangular board or grid of cells, where each cell has eight immediate neighbours (north, south, east, west, northeast, northwest, southeast, southwest). In a particular *generation*, each cell is either *alive* or *dead*. The state of each cell in the next generation is determined by simple pre-set rules that take into account the state of the cell and its eight immediate neighbours in the current generation, with death as a result of loneliness (not enough neighbours) or overcrowding (too many neighbours).

You are supplied with an implementation of this game in "core" ML, see the course web page. Your task is to design a modular implementation using signatures, structures and functors, by splitting this code into modules, making explicit the necessary dependencies of some parts upon others. Little or no new code should be required to achieve this. Your aim should be to identify re-usable entities while also encapsulating information. Each structure (functor) should have an explicit (output) signature. Where appropriate, opaque signature ascription is preferable to transparent signature ascription.

The modular structure of the resulting system should be such that it is easy to change various aspects of the game, the representation of data, and the algorithms used. For instance, it should be possible to change the shape and/or representation of the board without rewriting the algorithm used to compute the next generation. Possible board shapes include: a flat grid, with or without edges; a finite or infinite cylinder; a torus (doughnut); a sphere; a hexagonal grid (honeycomb); a 3-dimensional grid. On boards with edges, some cells will have fewer neighbours than others. A board may be represented as an array of boolean values, a list of coordinates of the live cells, a function mapping coordinates to booleans, etc. Other aspects of the game that might change include: the rules for computing the next state of a cell (while assuming that the next state of a cell will always be fully determined by its current state and that of its neighbours); the definition of "neighbours"; the way that the board is displayed. You may make assumptions — for example, that if cell a is a neighbour of cell b then b is a neighbour of a — but these should be made explicit in the form of comments.

⋆

If you want to learn more:

- Experiment with making some of the changes suggested above.
- The game of life is a simple example of a cellular automaton. Learn about cellular automata; a good starting point is the article

  P. Sarkar. A brief history of cellular automata. *ACM Computing Surveys* 32:80–107 (2000)

  Can you extend your system to handle the general case?

  Another source of information about cellular automata is the controversial book

S. Wolfram. *A New Kind of Science.* Wolfram Media Inc. (2002)

which describes cellular automata as the key to most of the mysteries of the universe. Following this line, create artificial life and/or use your system to find the answer to life, the universe and everything. (Only kidding!)

- There is a lot of information about the game of life on the web. A good starting point is `http://www.radicaleye.com/lifepage/` which includes a catalogue of interesting patterns. See `http://rendell-attic.org/gol/tm.htm` for a Turing machine implementation, and `http://cafaq.cafaq.com/lifefaq/index.php` for a FAQ.

- Learn about the extensions to the Standard ML module language that Moscow ML implements (in particular: higher-order, first-class and recursive modules) — see chapter 11 of the *Moscow ML Owner's Manual* — and think about how they might be used in this case. (These features are not covered by this course; please *do not* use them in your solution to the exercise.)