Dept. of Computer Science, University of California, Davis
ECS30 Instructor: Rob Gysel 3/7/17

# Programming Assignment #7*
# Due date: 3/15/17 11:59pm

Programs are to be submitted to Gradescope by the due date. You may work alone or in groups of two. Programs submitted up to 24 hours late will still be accepted but incur a 10% grade penalty. Uploading your programs to gradescope will immediately score your submission.

**Your program grade will be the score of the *last* submission that you have uploaded.**

Programs must compile using `gcc -Wall` without any warnings. Each program that compiles with a warning will incur a 10% grade penalty.

In this program, you will create a *cellular automata* called Brian's Brain. This cellular automata consists of squares (cells) on a 2D grid that are in one of three possible states: `ON`, `OFF`, or `DYING`. At each step, squares change their states depending on their neighbors. This assignment will guide you through this process, but you may be interested in the following outside resources on Brian's Brain:

**Wikipedia:** A brief description is at: `https://en.wikipedia.org/wiki/Brian's_Brain`

**YouTube video:** Animated generations are at: `https://youtu.be/1gKPsk9YHl0?t=11s`

**Interactive simulator:** You will create a command-line version of this: `https://scratch.mit.edu/projects/13866913/`

## Assignment instructions

Proceed by moving from the first part to the last part, in order. You must modify the supplied code. You do not need to define functions, and only need to add code to implement certain functions throughout the code. These are marked with:

```
// TODO: complete this function
```

To assist with your programming, the following testing files have been provided: `list_tester.c`, `cell_tester.c`, and `cell_grid_tester.c`. You may use these to compile executables that will test your code and give you feedback. Note that incorrect implementations may make these tests crash. A sample executable is available on the CSIF, and you can copy it to your current directory using this command:

*Last updated March 7, 2017

```
cp  /home/rsgysel/brians_brain_cellular_automata .
```

You can run it by giving it a *seed file* (see the files included for this assignment), and a number of generations to run, like this:

```
brians_brain_cellular_automata oscillator.txt 100
```

This runs the cellular automata using `oscillator.txt` for 100 generations.

Files to turn in:

```
brians_brain.c
brians_brain.h
cell_grid.c
cell_grid.h
Makefile
cell.c
cell.h
list.c
list.h
```

## Part 1: `cell.c` (10 pts, 2 test cases)

The `Cell` type is defined as a `struct` in `cell.h`. This represents a single cell (box) on the 2-dimensional grid. Each `Cell` has three members: an x-coordinate, a y-coordinate, and a `CellState` (one of `ON`, `OFF`, or `DYING`). The x and y-coordinate specify the `Cell`'s position on the grid. Each cell has 8 neighbors, the 4 cells adjacent above, below, left and right of it, and the 4 cells adjacent diagonally.

1. Implement `bool Cell_AreNeighbors(Cell C1, Cell C2)`.

## Part 2: `list.c` (10 pts, 2 test cases)

1. Implement `void List_Print(List* list)`. Hints: use a `for` loop to iterate through the nodes of the list. Use the `List` member `head` for the setup of your for loop, the `ListNode` member `next` for the increment of your for loop, and you should compare with `NULL` for the test case of your for loop.

## Part 3: `cell_grid.c` (30 pts, 6 test cases)

The `CellGrid` type is defined as a `struct` in `cell_grid.h`. This represents the 2-dimensional grid of cells. At each phase (generation) of the simulation, all `Cells` in the `CellGrid` will be in one of the three states `ON`, `OFF`, or `DYING`. In Part 4, we use the current generation's (current `CellGrid`) states to compute the next generation (a new `CellGrid`).

1. Implement `CellGrid* CellGrid_Create(int numRows, int numCols)`. This should use `malloc` to allocate a `CellGrid`.

2. Implement `void CellGrid_Delete(CellGrid* G)`. This should use `free` to de-allocate a `CellGrid`.

3. Implement `CellState CellGrid_GetState(const CellGrid* G, int row, int col)`. This function returns the state of the cell specified by the input row and column.

4. Implement `void CellGrid_SetCell(CellGrid* G, Cell C)`. This function sets `C` as a `Cell` of `G`. Note that `C` has x and y-coordinates that need to be used to specify which square of `G` to modify.

5. Implement `bool CellGrid_Inbounds(const CellGrid* G, int row, int col)`. This checks whether or not the input row and column is in the grid `G`.

6. Implement `void CellGrid_Print(const CellGrid* G, FILE* fp)`. This prints all of the cells according to their `CellState`. To begin, print the state of cell (0,0) first (in the top left corner) and then the rest of that row (all cells (0,j) where j is in bounds) before printing a new line and then proceed with the remaining lines.

## Part 4: `brians_brain.c` (30 pts, 6 test cases)

In Brian's Brain, the grid starts in an initial configuration of cell states. You can use the seed files `oscillator.txt`, `diamond.txt`, `gliders.txt`, and `spaceship.txt` to specify the initial configuration and run the program, like this:

```
brians_brain_cellular_automata oscillator.txt 100
```

This runs `brians_brain_cellular_automata` on `oscillator.txt` for 100 generations (generations are described below).

Seed files consist of a number of rows and columns on the first line, and on each line afterwards a row, a column, and a state (either 'O' for `ON` or 'D' for `DYING`, otherwise a `Cell` is assumed to be `OFF`).

Brian's brain runs for a finite number of `generations`. Each generation $t$ completely determines the states for the `Cell`s in generation $t + 1$ according to the following rules:

1. If a `Cell` is `ON` in generation $t$, then it is `DYING` in generation $t + 1$.

2. If a `Cell` is `DYING` in generation $t$, then it is `OFF` in generation $t + 1$.

3. If a `Cell` is `OFF` in generation $t$ and has exactly 2 neighbors that are `On` in generation $t$, then it is `ON` in generation $t + 1$.

1. Implement `CellGrid* NextGeneration(CellGrid* generation)`. This calculates a new generation from `generation` using the rules described above.

2. Implement `List* GetNeighboringCells(Cell cell, CellGrid* generation)`. This should return a list of all cells that are adjacent to `cell`, except for `cell` itself. Use `List_Create` to allocate your `List` and use `List_PushFront` to add elements to your `List`.

## Part 5: `Makefile` (10 pts)

Create a `Makefile` to create your project. Your makefile should have the following targets.

**cell.o** Builds the `cell.o` object file from `cell.c`.

**list.o** Builds the `list_tester` object file from `list.c`.

**cell_grid.o** Builds the `cell_grid.o` object file from `cell_grid.c`.

**brians_brain.o** Builds the `brians_brain.o` object file from `brians_brain.c`.

**cell_tester** Builds the `cell_tester` executable.

**list_tester** Builds the `list_tester` executable.

**cell_grid_tester** Builds the `cell_grid_tester` executable.

**brians_brain_cellular_automata** Builds the `brians_brain_cellular_automata` executable.

**all** Creates the tester executables and the main program, `brians_brain_cellular_automata`.

**clean** Deletes all *.o object files and executable files