CMSC 15100 **Introduction to Computer Science** **Project 1**
Autumn 2009 **November 9**

Snake

Due: November 16; 10pm

Introduction

Your task in this project is to implement the interactive game Snake. The player controls the direction of the snake using the arrow keys as it continuously moves over a board. the goal is to eat food, which makes the snake bigger. If the snake hits the wall or itself, then it dies and the game ends.

You will use the `world.ss` teachpack for this project and you should follow steps outlined below. Be sure to test your code for each step before moving onto the next step.

Step 1 — the game board

The playing area (or board) is divided up into square cells. We refer to a cell by its row and column ($\langle r, c\rangle$), with cell $\langle 0, 0\rangle$ being the upper-left corner of the board. The position of the center of the cell $\langle r, c\rangle$ will be $\langle wc + \frac{w}{2}, wr + \frac{w}{2}\rangle$, where w is the width (in pixels) of a cell. You should define constants for the board dimensions and the pixel size of the board cells. We recommend a board size of around 40×40 with a cell size of around 10 pixels.

```
(define num-rows ...)
(define num-cols ...)
(define cell-sz ...)
...
```

Step 2 — data structures

The second step is to define the data structures that are used to represent the state of the game.

```
;; a cell is specified by its row and column, which are numbers
(define-struct cell (row col))

;; a direction is one of 'left, 'right, 'up, or 'down

;; a board is a snake, a list of food cells, and the number of ticks
;; the board should satisfy the invariant that the snake segments do
;; not overlap with the food and that there is at most one piece of
;; food per cell
(define-struct board (snake food ticks))
```

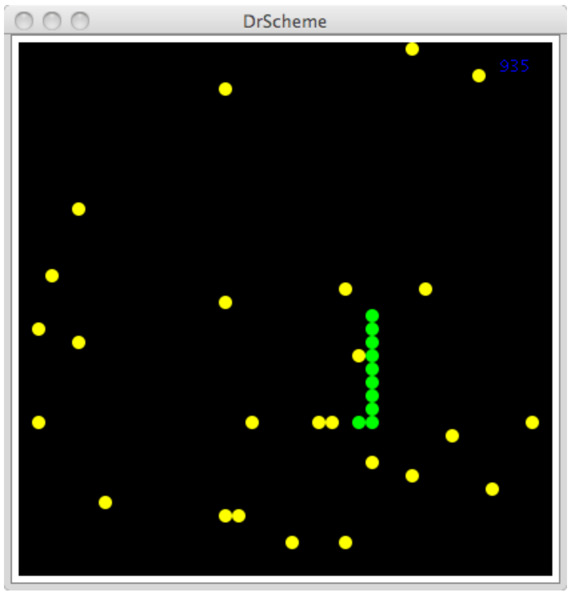

Figure 1: A screen shot of the game in progress

```
;; a snake is ...
```

For the snake, you need to keep track of its direction, the list of cells that make up its body (called *segments*), and whether it is alive or not.

Step 3 — rendering the board

The first function that you write should render the world graphically as shown in Figure **??**.

```
;; render-board : board -> scene
;; render the board as a scene.
(define (render-board w) ...)
```

The board should be black and the food and snake segments are rendered as circles. The food is yellow and the snake is green when it is alive and red when it is dead. We also report the current score in the upper-right corner. The scoring function is

```
;; compute the score of the board
;; score : board -> number
(define (score b)
  (- (* 100 (snake-length (board-snake b))) (board-ticks b)))
```

where `snake-length` is a function that returns the current length of the snake (*i.e.*, number of segments).

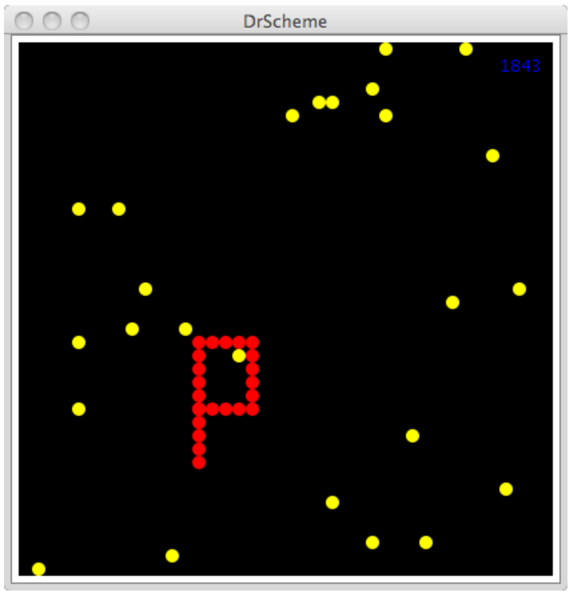

Figure 2: A screen shot of the game with a dead snake

Step 4 — updating the board

The most complicated part of the implementation is the code for updating the board. You will define a function that advances the board by one time step.

```
;; tick : board -> board
;; advance the state of the board b by one time step
(define (tick b) ...)
```

When the snake is alive, this function must do the following things:

- increment the ticks field of the board
- move the snake one cell
- detect if the snake has run into the wall or itself, in which case the snake dies.
- if the snake lands on some food, then the snake gets longer by one segment and replacement food is added to the board.

New food should be placed randomly, but it must avoid cells that are already occupied by either the snake of other food.

```
;; place-food : list-of-segments list-of-food-cells -> list-of-food-cells
;; randomly place a new piece of food, while avoiding cells that are
;; already occupied by the snake segments or food.
(define (place-food segs food) ...)
```

A random cell can be generated using the expression

```
(make-cell (random num-rows) (random num-cols))
```

When the snake moves, it gains a new segment at the front and loses a segment at the end (unless it is eating). You may find it useful to define a helper function that returns the neighbor of a cell in a given direction.

```
;; neighbor : cell direction -> (union cell false)
;; give a cell and a direction, return the neighbor in that direction.
;; If the neighbor is off the board, then return false.
(define (neighbor cell dir) ...)
```

Step 5 — handling player input

The player will use the arrow keys to control the direction of the snake. These are represented as the symbols `'left`, `'right`, `'up`, and `'down`. You will write a function that updates the board based on these inputs; any other input should be ignored.

```
;; update : board symbol-or-char -> board
;; update the board state based on the player command cmd
(define (update b cmd) ...)
```

Step 6 — running the game

To run the game, we will use the `big-bang` function from the `world.ss` teachpack to define a `run` function.

```
;; run : number -> true
;; run the game with the specified amount of food.
(define (run food-amount)
  (and
   (big-bang board-wid board-ht 1/8 (new-board food-amount))
   (on-redraw render-board)
   (on-tick-event tick)
   (on-key-event update)
   (stop-when (lambda (b) ...))))
```

You will need to fill in the argument to the `stop-when` function and define the `new-board` helper function that creates an initial board. We recommend that you place the snake in the middle of the board facing `'up` and randomly place the food particles.

Hints and guidelines

Here are some hints and guidelines for the project

- Do not leave this project to the last minute!
- Use library functions. The PLT Scheme system defines many useful functions (especially on lists) that you can use to reduce your coding effort.
- Do not leave "magic numbers" (*i.e.* constants) floating around your code. These should be defined at the top of the file.

- Each step described above corresponds to a logical part of the system that you are implementing. You should organize your code into sections based on this outline, with the main function (*e.g.*, `render-board` and `tick`) of each section presented first. Put the test cases for each section last.
- Make sure your code is commented and clear.