# Project 1: Snake

Due: November 12th

Your goal is to implement a variation of the game Snake. In this game, you control a snake that wanders around a square board, eating food. Each time the snake eats a morsel of food, it gets a little bit longer. If the snake ever runs into itself or runs into the edge of the board, the game is over. The snake never stops moving; the player can control only the direction that the snake moves.

For this project, use the `world.ss` teachpack. Start with this `world` data definition:

```
;; a world is:
;; (make-world snake food number)
(define-struct world (snake food ticks))

;; a snake is ...

;; a food is ...
```

and design data to represent snakes and their food. The snake should have three properties: the location of its body segments, the direction it is heading, and whether or not it has died.

Develop these functions (and any necessary helper functions):

- `render-world : world → scene`

  This function should render a world graphically, for example as shown in figure 1. The rendering should show all of the locations of food, the location of the snake, whether or not the snake is dead, and the score. In this picture, the yellow dots indicate food that is on the board, the snake is colored red to indicate it is dead, and the score is shown in the upper right-hand corner.

- `tick : world → world`

  This function should take one step forward in time for the world. It should

  - increment the `ticks` field of the world,
  - move the snake unless the snake is dead,
  - detect if the snake has run into itself or a wall, and
  - eat food (make the snake longer, remove the eaten food, and generate a new piece of food) when the snake lands on some food.

  Use the `random-posn` function to produce a location for a new piece of food (and remove the eaten piece of food from the world):

  ```
  ;; random-posn : number number number -> posn
  ;; to invent a new location for a piece of food on a board
  ;; of size (w,h) where the snake's segment size is cell-size
  (define (random-posn w h cell-size)
    (make-posn (* cell-size (random (/ w cell-size)))
               (* cell-size (random (/ h cell-size)))))
  ```

  Note that moving the snake means that the snake both gains and loses a segment (unless it eats). The new segment's coordinates are determined by the segment that was previously at the front of the

snake and the direction the snake is facing. The snake loses the oldest segment, namely the one that was previously at the end of the snake.

For example, figure 2 shows the snake before and after a single tick when the snake is moving upwards without eating. Figure 3 shows the same snake before and after a single tick when the snake eats a piece of food.

- `change : world symbol-or-char → world`

  This function should adapt the world based on keyboard input. The only keyboard input you are concerned with here are the arrow keys. If the user pushes one of the arrow keys, the second argument to `change` will be one of the symbols: `'left`, `'right`, `'up`, or `'down`. When one of those keys is pressed, the snake's direction should change (but nothing else).

Use this scoring function (although you may adjust the `25` to a number that is appropriate for your board size and amount of food):

```
;; score : world -> number
;; to compute the score for this world
(define (score w)
  (- (* 25 (snake-length w))
     (world-ticks w)))

;; snake-length : world -> number
;; to compute the length of the snake
(define (snake-length w) ...)
```

Only after you have completed the entire design recipe process for these functions (including testing them!), supply them to the `world.ss` teachpack like this to play the game, filling in the ellipsis with an appropriate initial world:

```
(big-bang world-width world-height 1/8 (make-world ...))
(on-redraw render-world)
(on-tick-event tick)
(on-key-event change)
```

Some notes on organizing your program:

- Do not leave constants floating around in your code. Define them at the top of the file. For example, I expect you to define the size of the window and the size of each morsel of food and each segment of the snake at the top of the file and then draw based on those variables (plus possibly more).
- Organize your functions (in your file) around your data definitions. That is, group functions that operate on snakes together, group functions that work on food together, and group functions that work on worlds together.
- Also, be sure to present your code from the top-down. That is, show the high-level functions first, and then show functions that they call.
- Finally, put all of your test cases together at the end of each section (organized by the functions they test, of course). Put the tests for helper functions before the tests for the functions that use them (this will help you more easily track down bugs).

Overall, imagine someone (me!) is going to be reading your code and expecting it to be well-written. Organize your code so that it is clear what is going on, to the best of your ability.

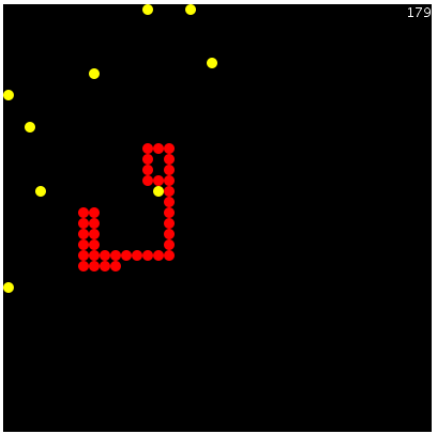


Figure 1: An example world, with lots of food and a dead snake

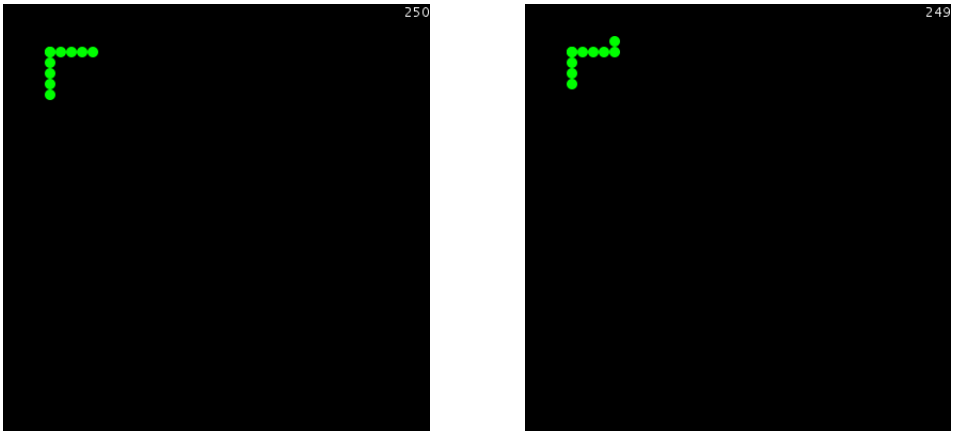


Figure 2: Before and after the snake moves upwards

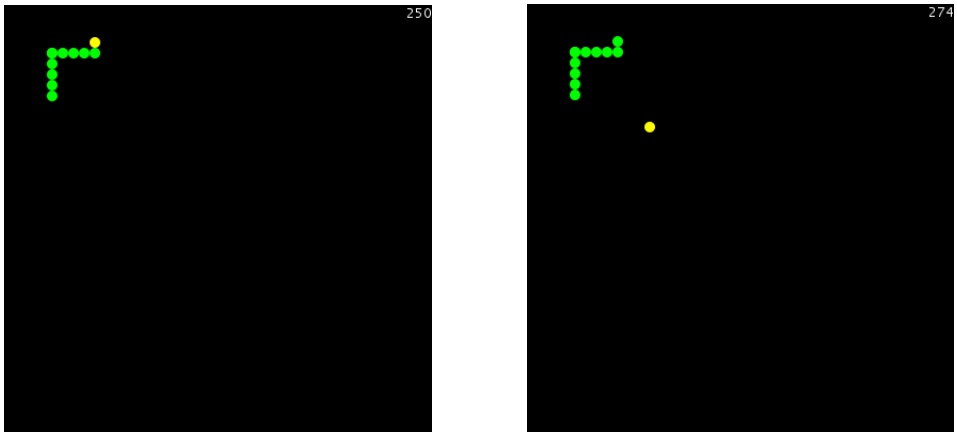


Figure 3: Before and after the snake eats (in a world with only a single piece of food)