Assignment:	11
Due:	Wednesday, April 14th at 11:45 pm Eastern Time
Language level:	Intermediate Student with lambda
Files to submit:	`ontario.rkt rl.rkt`

- Join the discussion for the answers to frequently asked questions.
- Unless stated otherwise, all policies from the previous assignment carry forward.
- This assignment covers material up to the end of Module 16.
- The **only** built-in functions and special forms you may use are listed below. If a built-in function or special form is not in the following list, you may not use it:

```
* + - ... / < <= = > >= abs add1 and append boolean? ceiling char<=? char<? char=?
char>=? char>? char? check-expect check-within cond cons cons? define define-struct
 eighth else empty? equal? even? exp expt fifth filter first floor foldl foldr
fourth integer? lambda length list list->string list-ref list? local map max
member? min not number->string number? odd? or quotient range remainder rest second
 seventh sixth sqr sqrt string->list string-append string-length string-ref
string<=? string<? string=? string>=? string>? string? sub1 substring symbol=?
symbol? third zero?
```


- Remember that basic tests are meant as sanity checks only; by design, passing them should not be taken as any indication that your code is correct, only that it has the right form.

- Unless the question specifically says otherwise, you are always permitted to write helper functions to perform *any* task. You may use any constants or functions from any part of a question in any other part.

- For any inexact tests, use a tolerance of 0.0001.

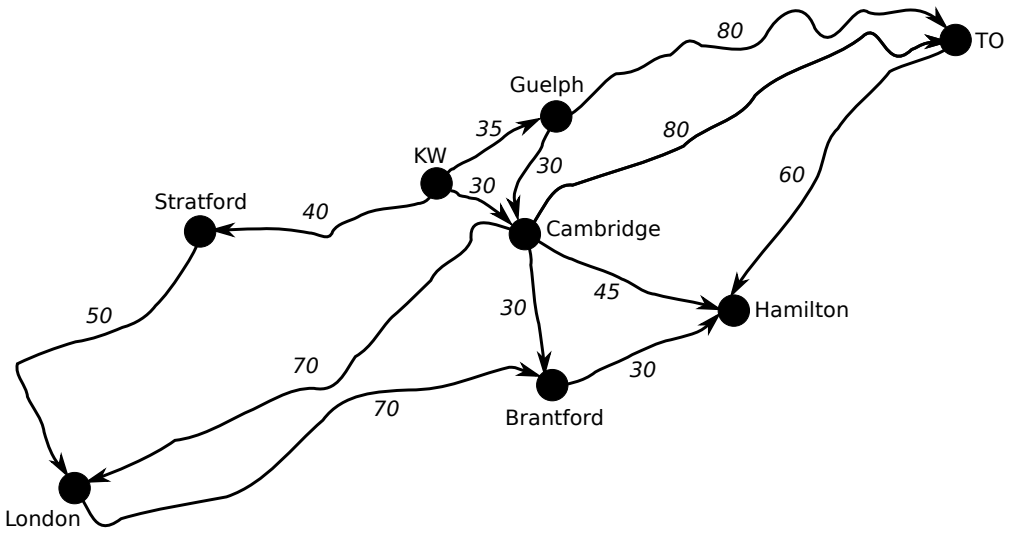
---

1. Pathing [50%]. Place your solutions to all parts of this problem in `ontario.rkt` Whenever we were using graphs so far, we attached values to vertices (vertex names in the form of a `Sym`) but we never gave edges values. By doing so, however, we can increase the usefulness of graphs significantly.

```
;; A Node is a Sym
;; A Map is a (listof (list Node (listof Neighbour)))
;;   requires: Map is directed and acyclic
;; A Neighbour is a (list Node Nat) where the number indicates the
;;   travel time (in minutes) to the neighbour.
```

. . . .

1.1. . For example, consider the map of Southern Ontario shown below. The values on each edge indicate travel times (in minutes) between nodes. With its help you could calculate your estimated travel time between different cities.



Exercise

Write a function `travel-time`. This function consumes an origin and a destination in the form of two `Sym` as well as a **`Map`**. It then produces the travel time between origin and destination; if no path exists, it produces false. `travel-time` requires that both origin and destination are on the map. (If there are multiple paths between the origin and destination, any correct travel time is acceptable.)


```
(define southern-ontario
  (list (list 'Brantford (list (list 'Hamilton 30)))
        (list 'Cambridge (list (list 'Brantford 30) (list 'Hamilton 45)
                               (list 'London 70) (list 'TO 80)))
        (list 'Guelph (list (list 'Cambridge 30) (list 'TO 80)))
        (list 'Hamilton empty)
        (list 'London (list (list 'Brantford 70)))
        (list 'KW (list (list 'Cambridge 30) (list 'Guelph 35) (list 'Stratford 40)))
        (list 'Stratford (list (list 'London 50)))
        (list 'TO (list (list 'Hamilton 60)))))

(check-expect (travel-time 'Guelph 'Hamilton southern-ontario) 90)
; travel time for '(Guelph Cambridge Brantford Hamilton)
```

. . . .

1.2. . Depending on the implementation details for `travel-time`, the function might not have produced the shortest possible travel time between origin and destination. We want to write a function that calculates the travel times for all valid paths. For now, however, we will not concern ourselves with the travel times but will concern ourselves with considering *every* path.

Exercise

Write a function `all-paths`, that produces all valid paths between a source and a destination. `all-paths` consumes an origin and a destination in the form of two `Sym` as well as a `Map`. It then produces a list of all possible paths between origin and destination as a `(listof (listof Sym))`. `all-paths` requires that both origin and destination are on the map. (If there are multiple paths between the origin and destination, the order of these paths does not matter.)

```
(check-expect
 (all-paths 'Guelph 'Hamilton southern-ontario)
 (list (list 'Guelph 'Cambridge 'Brantford 'Hamilton)
   (list 'Guelph 'Cambridge 'Hamilton)
   (list 'Guelph 'Cambridge 'London 'Brantford 'Hamilton)
   (list 'Guelph 'Cambridge 'TO 'Hamilton)
   (list 'Guelph 'TO 'Hamilton)))

(check-expect (all-paths 'Stratford 'Guelph southern-ontario) empty)
```

. . . .

1.3. . Now that we have mastered map-traversal, we can produce more complex data.

Exercise

Write a function `all-travel-times`. It consumes an origin and a destination in the form of two `Sym` as well as a `Map`. It then produces a list of all possible paths between origin and destination as well as their travel time as a `(listof (list Nat (listof Sym)))`. As before, `all-travel-times` requires that both origin and destination are on the map. (If there are multiple paths between the origin and destination, the order of these paths does not matter.)


```
(check-expect
 (all-travel-times 'Guelph 'Hamilton southern-ontario)
 (list (list 90 (list 'Guelph 'Cambridge 'Brantford 'Hamilton))
   (list 75 (list 'Guelph 'Cambridge 'Hamilton))
   (list 200 (list 'Guelph 'Cambridge 'London 'Brantford 'Hamilton))
   (list 170 (list 'Guelph 'Cambridge 'TO 'Hamilton))
   (list 140 (list 'Guelph 'TO 'Hamilton))))

(check-expect (all-travel-times 'Stratford 'Guelph southern-ontario) empty)
```

---

2. Escaping a Maze [50%]. Place your solutions to all parts of this problem in `rl.rkt`

Reinforcement learning is a style of machine learning that mimics the way in which animals can learn through positive (or negative) reinforcement. The driving concept behind reinforcement learning is simple — an entity wants to repeat behaviour that it found beneficial, and avoid behaviour that it found penalizing or wasteful. For this question you will implement a reinforcement-learning agent that can solve a wide variety of problems when configured properly, which we will use to solve a maze. This question is rather involved and has some important particular specifications you must follow in order for the auto-marking scripts to work correctly, so follow instructions carefully or you will be unable to receive correctness marks!

Here are the major ideas behind reinforcement learning at a high level and how they correspond to our assignment:

- An **agent** is the program that is trying to learn how to solve a problem. In our program the agent represents the person trying to escape the maze.

- A **state** is the current situation the agent finds itself in. In our program the state is whatever position the agent is currently located at, represented by a list of two Nat's `(list Nat Nat)`.
- An **action** is something the agent can "choose" to "do" when in a given state. In our program the actions will be to move North, East, South, or West.
- A **reward** is a numeric value the agent receives, which they are programmed to maximize. In our program the agent will receive a reward of 1 when reaching the goal of the maze and a reward of 0 at all other times.
- A **Value Function** is a function that maps from State-Action pairs and produces a numeric value that represents how much the agent should "value" taking a particular action in a particular state. The "value" represents how likely that action is to lead to reward based on the agent's previous experiences. In our program the value function will be a piece of data with the following data definition:

```
;; An Action Map (ActionMap) is a:
;; (list (list 'north Num)
;;       (list 'east Num)
;;       (list 'south Num)
;;       (list 'west Num))

;; A State is a:
;; (list Nat Nat)

;; A Value Function (ValFn) is a:
;; (listof State ActionMap)
```


Note: this is a dictionary of `States`, where the value of each `State` is a dictionary that maps our actions to a corresponding value.

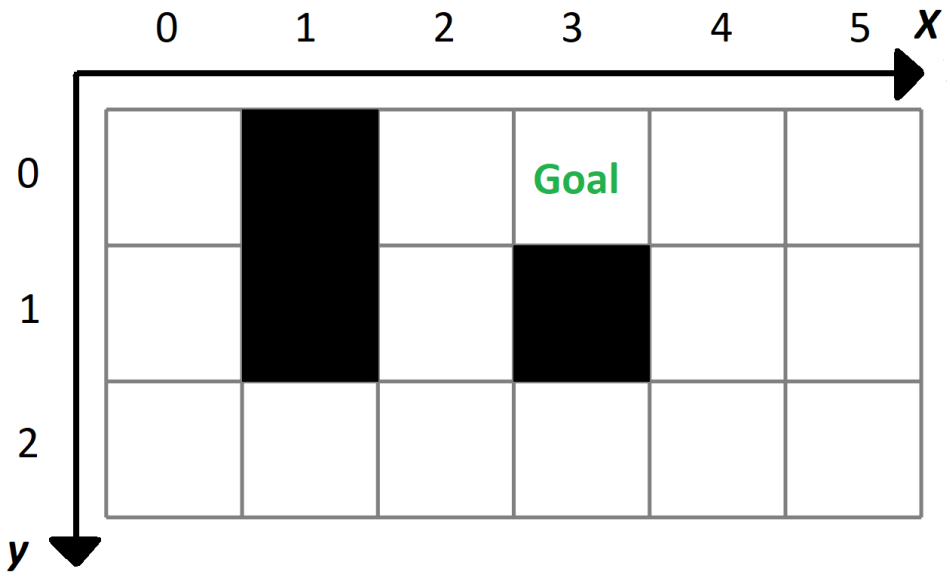
- An **episode** is a single instance of an agent taking a series of actions ending when it ultimately achieves its goal. Not all reinforcement learning problems are episodic. In our program an episode ends when the agent escapes the maze.

We will use the following data definition for a `Maze` which we will use as the problem our agent must solve:

```
;; A Maze is a (listof Str)
;; Requires: each Str be the same length, and contains only #\X, #\O, or #\G.

;; For example,
(define small-maze (list "OXOGOO"
                         "OXOXOG"
                         "OOGOOO"))
```

The `#\X` characters in the `Maze` represent walls which our agent cannot walk through, the `#\O` characters in the `Maze` represents open spaces which our Agent may walk into, and the `#\G` characters in the `Maze` represents *goal* square upon reaching which the agent receives a reward of 1 and the episode is complete. We specify `States` of our maze with an x and y coordinate, however our top left square is coordinate $(0,0)$ and our bottom right coordinate is $(w-1,h-1)$ where w is the width of the maze and h is the hieght of the maze. That is, the maze above can be represented by the following picture:



Note how the y-axis starts at 0 at the top and increases as we go *down*.

As this is a lot to take in we break the problem up into many small problems.

In order to make this possible, and to provide visualization of your agent, you are provided with the file `rl-support.rkt`. Download this file, and put it in the same directory as your solution. (Do not submit `rl-support.rkt`.)

At the top of the `rl.rkt` file that you created, write the following:

```
(require "rl-support.rkt")
```


You **do not** need to understand the contents of the file

Our functions will use "pseudo-random" numbers, which is to say fake randomness. In order to control this randomness so that test cases may be accurate you must follow our set of instructions exactly. For the sake of writing your own test cases we provide the function `set-randomness` that will be used to pre-set the randomness before generating numbers. `set-randomness` takes a single `Nat` as a parameter which we will call the *seed*.

. . . .

2.1. The Reward Function.

Exercise

Write the function `reward` that consumes a `Maze` and a `State` and produces the reward value of that `State` in the given maze. The reward should be 1 if the `State` corresponds to a goal state (`#\G`), and 0 otherwise. For example,

```
(check-expect (reward small-maze (list 0 0)) 0)
(check-expect (reward small-maze (list 3 0)) 1)
(check-expect (reward small-maze (list 1 2)) 0)
```

. . . .

2.2. Moving a Space.

Exercise

Write the function `move` that consumes a `Maze`, a `Sym`, and a `State` and produces a `State`. The `State` consumed represents the current location, and the `Sym` consumed must be one of `'north, 'east, 'south,` or `'west`. The produced `State` should be the result of moving from one position in the consumed direction. If there is a wall in the consumed direction, or moving in the consumed direction would take you out of the boundaries of the `Maze` then the produced `State` should be the original `State` consumed. For example,

```
(check-expect (move small-maze 'east (list 0 0)) (list 0 0))
(check-expect (move small-maze 'south (list 4 1)) (list 4 2))
(check-expect (move small-maze 'east (list 5 2)) (list 5 2))
(check-expect (move small-maze 'west (list 5 2)) (list 4 2))
```

. . . .

2.3. Producing our States. Our agent will ultimately make decisions based on the value function we define for them, and as such initializing it is very important! We will be initializing our value function with very small negative numbers, so that initially the Agent views every state as slightly negative — this is because they want to escape as quickly as possible and do not know anything about their environment yet so any movement that doesn't lead to escape is a waste of time!

This question will require you to follow a precise specification of how to use the provided functions, or else you will not be able to receive correctness marks.

Exercise

Write the `maze-states` that consumes a `Maze` and produces a `(listof State)` that is a list of each state in the `Maze`. Your produced `list` must order the `States` first on their `y` position, and second on their `x` position. That is a `State (list x y)` comes before a `State (list p q)` if `(< y q)`, or if `(= y q)` then if `(< x p)`. For example,


```
(check-expect
 (maze-states small-maze)
 (list (list 0 0) (list 1 0) (list 2 0) (list 3 0) (list 4 0) (list 5 0)
       (list 0 1) (list 1 1) (list 2 1) (list 3 1) (list 4 1) (list 5 1)
       (list 0 2) (list 1 2) (list 2 2) (list 3 2) (list 4 2) (list 5 2)))
```

Note the ordering of the `States`.

. . . .

2.4. Initializing our value function.

Exercise

Write the function `init-value-fn` that consumes a `Maze` and produces a `ValFn`. Because the `ValFn` is initialized randomly you must follow an important set of instructions to use the functions `init-state`, `set-randomness`, and `check-init` which are provided in `rl-support.rkt`. `init-state` consumes a `State` and produces the corresponding `(list State ActionMap)`. You must apply the `init-state` function to each state **exactly once**, and each application of `init-state` must be done to the `States` in the order they would appear if produced with `maze-states`. Additionally, your produced `ValFn` must be ordered the same way based on the `State` keys.

Your test cases for `init-value-fn` must look a little different due to the pseudo-randomness. Each `check-expect` must first locally define a constant named `random` to the value produced by an application of `set-randomness` applied to a `Nat seed`, then you should apply the provided `check-init` function with the arguments of (1) the result of applying your `init-value-fn` to your `Maze` and (2) the same `Nat seed` that was used to define `random`. `check-init` will produce `true` if your `init-value-fn` was correct, so the second argument supplied to your `check-expect` should be `true`. For example,

```
(check-expect
 (local [(define random (set-randomness 42))]
   (check-init (init-value-fn small-maze) 42)) true)
(check-expect
 (local [(define random (set-randomness 17))]
   (check-init (init-value-fn small-maze) 17)) true)
```


Note: this `check-expect` is comparing the `Bool` produced to `true`, so if you fail a test case you will only be told it is because `true` wasn't `false`, which isn't particularly useful for finding out any errors you made. As such you should use the output of the provided `first-n-inits` function, which consumes two `Nats n` and `seed` where `n` is how many values to produce and `seed` which should match the `seed` natural used in the test case. This will generate the first `n ActionMaps` that should appear in your solution. For example, `(first-n-inits 2 42)` would be a list of two `ActionMaps`, the first of which should correspond to the `ActionMap` for `State (list 0 0)`, and the second of which should correspond to the `ActionMap` for `State (list 1 0)` (as those should be the first two `States` which have their value produced).

. . . .

2.5. Choosing our Next Move. We must provide a decision policy for how our agent chooses its next action. The decision policy we will follow is the *epsilon-greedy* policy. The behaviour of the epsilon-greedy policy is that our agent will always take the action that they believe to have the highest value (act greedily), except for a small percentage of the time the agent will choose to "explore" and take a random action out of those available. The percent chance to explore is called the *epsilon* value, hence the name epsilon-greedy policy.

Again, since this function requires randomness you must take care to ensure your implementation follows the specification for how to use the provided helper functions `should-explore?` and `random-action`. You will also need the following definition of the list `actions` for use with `random-action`.

```
(define actions (list 'north 'east 'south 'west))
```

Exercise

Write the function `next-action` that consumes a `Maze`, `State`, a `ValFn`, and a `Num` and produces the `Sym` of the next direction the agent would like to move. The `State` represents the current `State` the agent is in, and the `Num` represents the epsilon value. Your function must first apply `should-explore?` **exactly once** to the consumed epsilon value. If the application of `should-explore?` is `true` then your function should apply the function `random-action` to the provided `actions` list **exactly once** and produce the value that the application produces. If `should-explore?` produced false then you should not have applied `random-action` at all, instead you should produce the action that has the highest value for the current `State`. For example,

```
(next-action tiny-maze (list 0 0) tiny-value-fn 0.05)
⇒ 'north if (should-explore? 0.05) produces false or (random-action actions) if it
    produces true
```

The above produces `'north` if it doesn't choose to explore because `'north` is the action because in `tiny-value-fn` defined below it is the action with the highest corresponding value in the `ActionMap` for `State (list 0 0)`. Due to the randomness your test cases again will need to take a special form:

```
(define tiny-maze (list "OG"))

(define tiny-value-fn
  (list (list (list 0 0)
              (list (list 'north 0.03)
                    (list 'east 0.02)
                    (list 'south 0.01)
                    (list 'west 0.00)))
        (list (list 1 0)
              (list (list 'north 0.03)
                    (list 'east 0.025)
                    (list 'south 0.017)
                    (list 'west 0.033)))))

(check-expect
  (local [(define random (set-randomness 42))]
    (next-action tiny-maze (list 0 0) tiny-value-fn 0.05))
  (local [(define random (set-randomness 42))]
    (cond [(should-explore? 0.05) (random-action actions)]
          [else 'north])))
```


Note: you could alternatively evaluate the second `local` expression manually and substitute it manually in the `check-expect`, that is:

```
(local [(define random (set-randomness 42))]
        (cond [(should-explore? 0.05) (random-action actions)]
              [else 'north])) ⇒ 'north
;; If you evalute the above expression and find it produces 'north then you can simply
    write the check-expect
(check-expect
  (local [(define random (set-randomness 42))]
    (next-action tiny-maze (list 0 0) tiny-value-fn 0.05)) 'north)
```

Note: that each expression still must be wrapped in the `local` that defines `random` with the same `seed` value, or you cannot ensure they are corresponding values.

. . . .

2.6. Updating the Value Function. One of the most important things we must define for our agent is how it decides to value actions. Each time our agent takes an action and sees the result, it should update the

value of taking that action in the given `State`. The process for determining the new value of an action `A` in a given `State` `S` is as follows:

1. Pull the current value of taking action `A` in `State` `S` from the current `ValFn`. Let us call this value `Current-Value`
2. Determine the new state `S-new` that is the result of taking action `A` in `State` `S` using your function `move`
3. Determine the reward value, `r`, for entering `S-new` using your function `reward`
4. Produce the value of the most highly-valued action in `S-new`. That is, search the current `ValFn` for `S-new` and pull the highest value of the `ActionMap` for `S-new`. Let us call this value `Future-Estimate`
5. Calculate the new value for taking action `A` in `State` `S` as:
   `Current-Value` + `0.0125`*(`r` + 0.9*`Future-Estimate` - `Current-Value`)

Exercise

Write the function `new-value` that consumes a `Maze`, a `State`, a `ValFn`, and a `Sym` reprsenting an action, and produces the new value of the given `State`-action pair as per the procedure above. Your test cases should use `check-within`. For example,

```
(check-within
 (new-value tiny-maze (list 0 0) tiny-value-fn 'east)
 0.03262125
 0.0001)
```

Note how the calculated value was given by the calculation:

$$0.02 + 0.0125(1 + 0.9 \cdot 0.033 - 0.02)$$

Make sure you understand how the above values were calculated given the definitions of `tiny-maze` and `tiny-value-fn` above.

. . . .

2.7. Baby's First Steps. We have almost completely defined a reinforcement-learning agent! All that is left is to put together the pieces we've written to execute a single step of our agent's "life".

We will now be using your function `next-action` which used our provided functions. Again to account for the randomness we must follow a very specific set of instructions about how much to use the function `next-action`.

Exercise

Write the function `take-step` that consumes a `Maze`, `State`, `ValFn` and a `Num` and produces a `(list State ValFn)`. The `State` consumed represents the `State` the agent is currently in. The `Num` consumed is the epsilon value for the epsilon-greedy policy we follow for selecting actions. The `State` in the produced `list` is the new `State` the agent is in after choosing and taking a single action. The `ValFn` in the produced `list` is the result of updating the consumed `ValFn` so that the action we chose to take in the consumed `State` is updated with the value calculated with our `new-value` function. In order to manage the randomness your function `take-step` must call your function `next-action` **exactly once** to determine the action to take and call none of the other functions we've provided you. Again, due to randomness we must write special test cases. Additionally, since our `ValFn` contains inexact numbers you should use `check-within`. For example,

```
(check-within
 (local [(define random (set-randomness 42))]
   (take-step tiny-maze (list 0 0) tiny-value-fn 0.05))
 (list (list 0 0)
       (list (list (list 0 0)
                   (list (list 'north 0.0299625)
                         (list 'east 0.02)
                         (list 'south 0.01)
                         (list 'west 0.00)))
             (list (list 1 0)
                   (list (list 'north 0.03)
                         (list 'east 0.015)
                         (list 'south 0.017)
                         (list 'west 0.033)))))
 0.0001)
```


Note: the expected value for this test case was produced by first evaluating the expression:

```
(local [(define random (set-randomness 42))]
       (next-action tiny-maze (list 0 0) tiny-value-fn 0.05))
```


Taking note of the fact that it produced `'north` and then manually calculating what the new `State` and `ValFn` should be. You should write your test cases in a similar way. The value `42` can be any number, but should be the same for this test case and for your expression.

. . . .

2.8. They Grow Up So Fast. **This part is completely optional and just for fun!** So that you can see your beautiful reinforcement-learning agent in action we have provided a couple functions. First the function below:

```
;; run-n-episodes: (Step-Fn as Defined above) Nat (list Nat Nat) Maze ValFn Num -> (list
    ValFn (listof Nat))
(define (run-n-episodes step-fn n initial-state maze val-fn epsilon))
```

This function takes as its parameters your `take-step` function, a number of episodes you'd like to run, the state that your agent should begin at each episode, the Maze your agent should be stuck in, a value function for that `Maze` (likely the one generated with `init-value-fn`), and an epsilon value (we suggest starting with `0.05`). The function then runs your agent through a number of episodes equal to the value you passed in for `n`. Once all episodes have been ran the function produces a pair where the first value is the new value function your agent has learned, and the second value is a list of the number of steps your agent took to find the goal each episode. If you have configured your agent properly then the number of steps should get smaller **on average** each episode (on average because random chance has an affect on this, depending on how much your agent decides to explore). After enough episodes your agent should

generally escape the maze in the minimum number of steps, save for occasionally going a bit slower when they decide to explore.

The second function provided to you is:

```
;; run-graphic-maze: (Step-Fn as Defined above) (list Nat Nat) Maze ValFn Num -> GUI
(define (run-graphic-maze step-fn initial-state maze val-fn epsilon))
```

This function behaves similar to `run-n-episodes` but instead it produces a graphical display for you to watch your agent attempt to escape the maze. On the graphical display you will have to press the "Play" button to begin your agents journey. Unlike `run-n-episodes`, `run-graphic-maze` continually runs episodes until you close the display.

Since watching very early episodes of your agent learning a maze can be boring, and the graphical display will run slower than Racket can run without display graphics, we suggest the value function you provide to `run-graphic-maze` is one that you've already produced using `run-n-episodes` like so:

```
(define q-small (first (run-n-episodes take-step 3 (list 0 0) small-maze (init-value-fn
    small-maze) 0.05)))
```

Additional Notes. In the function `new-value` we used two values, `0.0125` and `0.9`. These two values should actually be parameters of our agent named the *learning rate* and *discount factor* respectively. To make it easier to describe the procedure of calculating the new value we gave you these values as predetermined, these values we've selected are not necessarily the best values you could use! If you want to have some fun after you're done the assignment, we suggest playing around with these values (as well as the value you use for epsilon) to find a more optimal solution that helps your agent learn the maze in as few episodes as possible.