Homework Assignment 1

Part II

When uploading to Gradescope, please ensure the filename is `main.rkt`, otherwise your assignment will get 0 points.

1. (5 points) In Part I.
   (a) (1.5 points) In Part I.
   (b) (3.5 points) In Part I.
2. (2 points) In Part I.
3. (7 points) *Your goal is to implement a Binary Search Tree (BST) in Racket, as we learned in Lecture 3 (user data-structures).*

   To this end, you will need to implement the constructor and selectors of each field, as well as the operation to *insert* a node in the BST. **Please use the function names declared in the homework assignment template, as otherwise you will get 0 points in this assignment.**

   The following code is a Java implementation of binary tree taken from the Wikipedia page on BST's[1].

```
class Tree {
  Tree left, right;
  float value;
  Tree(Tree left, float value, Tree right) {
    this.left = left;
    this.value = value;
    this.right = right;
  }
  Tree setLeft(Tree left) { return new Tree(left, this.value, this.right); }
  Tree setValue(float value) { return new Tree(this.left, value, this.right); }
  Tree setRight(Tree right) { return new Tree(this.left, this.value, right); }

  static Tree insert(Tree node, float value) {
    if (node == null) {
        return new Tree(null, value, null);
    }
    if (value == node.value) {
        return node.setValue(value);
    }
    if (value < node.value) {
        return node.setLeft(insert(node.left, value));
    }
    return node.setRight(insert(node.right, value));
  }
}
```

4. (9 points) *Your goal is to check if a datum is syntactically valid, with respect to the specification we introduced in Lecture 1 and Lecture 2.*

   Recall function `quote` from Lecture 3. This function produces a logical representation of the code given as parameter. The serialized code that results from `quote` is known as a *datum*, or a *quoted* term. In the following exercises, the quoted term shall **not** include boolean expressions and conditionals. A quoted expression will include numbers, `define`, `lambda`, and function application.

   For the sake of simplicity, there is no need to recursively check the syntactic validity (eg, you do not need to check the if the body of a `lambda` is syntactically valid). For instance, given a `lambda` are the parameters symbols? Does the body of a `lambda` has expected number datums as we discussed in class? You do *not* need to check the semantic validity of the datum (eg, check if a variable is defined).

[1] `https://en.wikipedia.org/wiki/Binary_search_tree`

(a) (3 points) Function `lambda?` takes a datum and returns a boolean whether or not the quoted term is a `lambda`.

```
(check-true (lambda? (quote (lambda (x) x))))
(check-false (lambda? (quote 3)))
```

You can check if a datum is a list of symbols with a combination of functions `symbol?`[2] and `andmap`:[3]

```
(check-true (andmap symbol? (quote (x y z))))
(check-false (andmap symbol? (quote (x 3 z))))
```

(b) (1 point) Function `lambda-params` takes a quoted lambda and returns the list of parameters (symbols) of the given function declaration.

```
(check-equal? (list 'x) (lambda-params (quote (lambda (x) y))))
```

(c) (1 point) Function `lambda-body` takes a quoted lambda and returns a list of terms of the given lambda.

```
(check-equal? (list 'y) (lambda-body (quote (lambda (x) y))))
```

(d) (1 point) Function `apply?` takes a datum and returns a boolean whether or not the quoted term is a function application.

```
(check-false (apply? (quote (lambda (x) x))))
(check-true (apply? (quote (x y))))
```

(e) (1 point) Function `apply-func` takes a quoted function application expression and returns the function being called.

```
(check-equal? 'x (apply-func (quote (x y))))
```

(f) (1 point) Function `apply-args` takes a quoted function application expression and should return the arguments (expressions) of the function being called.

```
(check-equal? (list 'y) (apply-args (quote (x y))))
```

(g) (0.2 points) Function `define?` takes a datum and returns a boolean whether or not the quoted term is a `define`.

```
(check-true (define? (quote (define x 3))))
```

(h) (1 point) Function `define-basic?` takes a datum and returns a boolean whether or not the quoted term is a *basic* definition, according the specification in Lecture 2.

```
(check-true (define-basic? (quote (define x 3))))
```

(i) (2.8 points) Function `define-func?` takes a datum and returns a boolean whether or not the quoted term is a *function* definition, according to the specification in Lecture 2.

```
(check-true (define-func? (quote (define (x) 3))))
```

[2] `https://tinyurl.com/yblyxmoz`
[3] `https://tinyurl.com/y7kv2mzt`