CSCI 275

Lab 01: The Basics

Due Thursday February 13 at 11:59 PM

In this lab you will create programs which put to use some of the week's topics. You'll also be introduced to some useful new ways of thinking about Scheme code. By the end of the assignment, you should be comfortable with

- The Dr. Racket Scheme interpreter
- Writing Scheme functions
- Using recursion with lists

Your solutions to the following exercises should all be placed in a single Scheme file with name *lab1.rkt* The first line of *your* file should be

```
#lang racket
```

Use Handin to submit your solutions.

Part 1 - Lists of Atoms

1. Some of the most intuitive Scheme functions work with lists of atoms such as the list (2 3 4). In this part we will write some of these functions. First, we need to rectify an omission. Some versions of Scheme have a primitive function atom? that returns *#t* if its argument is an atom and *#f* if it isn't; this isn't built into Racket. Since Racket has a primitive *list?* that returns *#t* if its argument is a list--i.e, either null or the result of a cons operation-- we can easily write *atom?*. Every expression is either an atom or a list. Use this to define function *atom?*. Here is some test data:
    - *(atom? 3)* returns *#t*
    - *(atom? '(1 2))* returns *#f*
    - *(atom? null)* returns *#f*

2. Write the *lat?* function from *The Little Schemer*. This takes an argument and returns *#t* if it is the empty list or a list whose every element is an atom.

3. Write the function *not-lat?* that returns *#t* if its argument is NOT a list of atoms. Of course, you could write this as *(define not-lat? (lambda (s) (not (lat? s))))*, but write it directly using *cond*.

4. Write the function *list-of-ints?* that returns *#t* if its argument is empty or if its argument is a list, each of whose entries is an integer. You can use the primitive function*integer?* for this.

5. Compare functions *lat?* and *list-of-ints?*. The structures of these functions should look very similar. Write function *list-of-same?* that takes two arguments: a predicate (which tests a condition) and a list. *(list-of-same? kind-of-element s)* returns *#t* if *s* is empty or if every element causes kind-of-element to return *#t*.
   (list-of-same? atom? s) should be the same as *(lat? s)*, and *(list-of-same? integer? s)* should be the same as *(list-of-ints? s)*.

6. Now rewrite *list-of-same?* as *list-of-same2* so that it takes only one argument, a predicate, and returns a function that takes an argument and says if the predicate returns *#t*for each element of the argument. Now *(list-of-same2 atom?)* is the same as *lat?* and *(list-of-same2 integer?)* is the same as *list-of-ints?* This process of taking a function of two arguments and rewriting it as a function of one argument that returns another function of one argument is called *currying* the original function, named after Haskell Curry, an important American mathematician who worked on the foundations of logic and programming languages.

Part 2: Operations on Lists

7. Write (allmembers lat1 lat2), which returns #t if every member of lat1 is also a member of lat2, and #f if that isn't true.
   - (allmembers '(a c x) '(a b x c x d)) returns #t
   - (allmembers '(a c x) '(a b c)) returns #f
   - (allmembers '(a) '( )) returns #f
   - (allmembers '( ) '( )) returns #t

8. Write (rember2 a lat), which removes the *second* occurrence of a from lat, if there is one.
   - (rember2 'x '(a b x c x d)) returns (a b x c d)
   - (rember2 'x '(a b x c x d x e)) returns (a b x c d x e)
   - (rember2 'x '(a b x c))) returns (a b x c)

9. Write (rember-pair a lat), which removes every occurrence of two consecutive instances of a in lat:
   - (rember-pair 'a '(a a b b c c a b c a a)) returns (b b c c a b c)
   - (rember-pair 'a '(a b c a b c a)) returns (a b c a b c a)
   - (rember-pair 'b (a b b b a)) returns (a b a)
   - (rember-pair "b (a b b b b a)) returns (a a)

10. Write (duplicate n exp), which builds a list containing n copies of object exp.

    - (duplicate 3 'x) returns (x x x)
    - (duplicate 0 'y) returns ()
    - (duplicate 3 '(a b c)) returns ((a b c) (a b c) (a b c))

11. Write (largest lat) where lat is a list of numbers. Naturally, this should return the largest value in lat.

- (largest '(4 6 3 4 5 1 2) returns 6

12. Write (index a lat) which returns the 0-based index of the first occurrence of atom a in lat. If a is not an element of lat this returns -1.
    - (index 'x '(x y z z y)) returns 0
    - (index 'y '(x y z z y)) returns 1
    - (index 'a '(x y z z y)) returns -1
    - (index 'x '( )) returns -1