Tutorial 6

Start by opening the file Tutorial 6.rkt.

Part 1

Look at the procedure **min-max-recursive-list** and the test case for it. If you run the code, you'll notice that the test case doesn't provide complete code coverage (some of the code is still black). Write another test case that will give you complete code coverage. That is, add a second test that makes sure that the black section of code gets run (and therefore tested).

Part 2

1. Use for-each and recursion to write an imperative procedure for computing the sum of all the numbers in a recursive list. Do not use fold, apply, or map.

2. Write an imperative procedure, (**fold-and-map** folder start mapper list), that computes the equivalent of (foldl folder start (map mapper list)) using for-each, rather than foldl or map. For example, (fold-and-map + 0 abs '(1 -2 3 -4)) should sum the absolute values of all the elements of the list, yielding 10.

3. Now write an imperative procedure that takes a recursive list of numbers, possibly containing duplicates, and returns a flat (non-recursive) list of all the numbers, with the duplicates removed. It can return the numbers in any order, but it should remove the duplicates.

   Hint: remember that you can use (member x list) to test whether x appears in list.

Part 3

Remember our social network example from Tutorial 3? We represented a network as a list of pairs of friends:

```
(define friends-db
  (list (list "ben" "jerry")
        (list "martha" "paul")
        (list "hillary" "bernie")
        (list "lizbeth" "mikael")
        (list "edward" "bernie")
        (list "steve" "hillary")
        (list "larry" "edward")
        (list "sheryl" "hillary")
        (list "sheryl" "martha")
        (list "lizbeth" "edward")
        (list "elliot" "lizbeth")
        (list "edward" "elliot")
```

```
(list "lizbeth" "berkoff")
(list "berkoff" "nikita")))
```

Most of these people are connected, in the sense that you can get from one to the other via a sequence of friendship relations. For example, you can get from Bernie to Steve by way of Hillary, and from Nikita to Elliot by way of Berkoff and Lizbeth. On the other hand, you can't get from Ben to Martha at all.

The connected component of a person is the set of people who are reachable from them via an arbitrary sequence of friendship relations. Write an imperative procedure to find the connected component of a person:

- Your procedure will need a local variable to hold the growing list of people in the social network, and
- A recursive helper procedure that takes a person as an argument and adds them to the list of people, then uses for-each to check the list for all the friends of the person, calling the helper recursively on each friend. HOWEVER, if the person already appears in the list, the helper procedure should do nothing (since they're already in the list.

For purposes of testing, it's useful to notice that Ben and Jerry have no friends except each other, so they are their own connected component. So (connected-component "ben") should return a list with just "ben" and "jerry" (in whatever order). But Lizbeth's connected component is much bigger.

Part 4 (advanced students)

One of the things that's nice about languages like Racket is that they let you write your own looping constructs. That is, map and foldl aren't built-in parts of the language, they're normal procedures you can write. Whereas in languages like Java, you're stuck with the constructs like while and for that the language gives you, and you can't write new ones. But in Racket, you can write while and for as normal code. The simple way of writing them is admittedly ugly. But on Friday, Spencer will talk about a new feature, called macros, that allows you to write more elegant versions of for and while.

1. Write a procedure, (**while** *test body*) that takes two parameterless procedures, *test*, and *body*, and repeatedly runs body as long as test is true. For example, the code:

   ```
   (local [(define x 0)]
     (while (λ () (< x 10))
            (λ ()
              (begin (display x)
                     (set! x (+ x 1))))))
   ```

   should produce the output: 0123456789. (Remember that the procedure display just prints its argument).

   Again, apologies for the ugly gratuitous λ expressions. Using macros you can make a much cleaner version.

2. Now use while to write a procedure, (**for** start test next body), that acts like a for loop, i.e. it runs start, then runs test, and if the test is true, then it runs body and next, then runs test again. So the equivalent of the code above using for would be (the admittedly very ugly) code:

```
(local [(define x 0)]
  (for (λ () (set! x 0))
       (λ () (< x 10))
       (λ () (set! x (+ x 1)))
       (λ () (display x))))
```