CSSE 304 Assignment 12 Updated for Spring, 2016

This is an individual assignment. It is important that each person do this to get up to speed on the concepts contained therein, which will be very important to the interpreter project, and a major component of Exam 2.

Turnin: Bring a written or printed solution to the Session 21 class meeting. (**Monday, April 18 in Spring, 2016**).
Turn it in at the beginning of class if you want to get credit for this problem.
No late days may be earned or used for this problem.

1. (30 points) Draw a "environments and closures" diagram like the ones we did in class, showing all of the closures and local environments created by execution of the following code, following the style used in the class examples, as well as the relationships between them.
   - A closure has **three parts** (argument list, code, environment pointer).
   - A local environment has **two parts** (a table of variables and their values, and a pointer to an environment produced by enclosing code (if any).
   - Environment pointers always point to environments, never to closures.
   - The value of a variable in an environment is never an environment, nor is it code.
   - You can't have two arrows coming from the same "pointer location".
   - Place sequence numbers (start with 1) near each environment or closure that you draw and near each new entry in the global environment, to indicate the order in which these references are created during the execution of the code.
   - For simplicity in the case of `let`, you should pretend that `let` is executed directly (without translation into an application of `lambda`), so that all you need to show is the environment extension, rather than creation of a closure followed by the environment extension created by the application of that closure.
   - Show the changes to the global environment, and include those in your sequence numbering.

     [Hint: My solution introduces 5 local environments, 4 closures, and 2 changes to the global environment. Yours probably should do the same]

```
(define compose2
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

(define h
  (let ([g (lambda (x) (+ 1 x))]
        [f (lambda (y) (* 2 y))])
    (compose2 g f)))

(h 4)
```