Problem 1: Understanding mceval.scm

If you look through the metacircular evaluator (Sections 4.1 – 4.5, complete code available from mitpress.mit.edu/sicp/code/ch4-mceval.scm) you will notice that it has no higher-order functions, except a couple of uses of map in primitive-procedure-names and primitive-procedure-objects. These uses of map could easily replaced by recursive procedures.

This is nice: You could write mc-eval is a language without higher order functions, such as C.

My question is: what happens when you try to evaluate a higher-order function? You might consider how the composition function (lambda (f g) (lambda (x) (f (g x)))) is evaluated.

Problem 2: Modifying mceval.scm

In class we wrote let->combination to add the let special form to mc-eval. We also write a direct implementation of let based on looking at – and simplifying – how mceval evaluated the result of let->combination.

A bit of pseudo-code for or:

1. Add an abstract data type for the representation of or expressions. You need to write or? testing whether an expression is an or expression, or-exprs returning the expressions following the or, and make-or making an or expression from a list of expressions.

2. Write or->simpler-exp that implements the pseudocode above, rewriting an or expression to an expression that may still contain an or but no longer has an or at the outermost level. Since the value of a sub-expression may only be evaluated once, you should use let in your translation of or and use one of the
let implementations from class in your mceval. (One of these implementations appears in the class handout for section 4.2: see the schedule page for a link.)

Write a line to put into eval to process or expressions using or->simpler-exp.

Where in the body of eval can you put this line?

Put your answers to (1) and (2) into or1.ss for submission – do not include the entire evaluator, just the new or changed lines.

(3) Consider the behavior of or. Code that behavior directly as routine eval-or that may call eval on subexpressions, but never creates a more complex piece of syntax. Remember that an expression after an or may only be evaluated once.

Put your eval-or and the line to use it from eval into or2.ss.

**Problem 3:** Exercise 4.8, page 376

Add “named let” to mceval by transating it into expressions that eval can already evaluate. Turn in your new or modified procedures in namedlet.ss. Even if you have only one line in eval for let you will need to write named-let?, named-let-bindings, and named-body because the syntax of named let is different from the syntax of regular let.

Put your new and modified code in named-let.ss

**Problem 4:** Exercise 4.22, page 398

Extend the analyzing evaluator from mitpress.mit.edu/sicp/code/ch4-analyzingmceval.scm to support the special form let.

This is not a large modification from the code for let in the class handout for section 4.2.

Turn in your new and modified code in analyzing-let.ss

**Problem 5:** (Extra credit)

Extend the non-deterministic evaluator from mitpress/sicp/code/ch4-ambeval.scm with the and special form as described in Exercise 4.4.

Turn in your new and modified code in ambeval-and.ss

The tough part of this problem is the code to execute if the evaluator fails back into the and.