Instructions: For this exercise you are encouraged to work in groups of two so that you can discuss the problems, help each other when you get stuck and check your partners work. This lab explores several variants of the metacircular interpreter used in the previous lab. In particular, students will study the lazy (normal order) interpreter including the variant that

1) lazymceval.rkt
2) lazymemmceval.rkt
3) ambeval.rkt

All files can be loaded into DrRacket and run. Executing the command (driver-loop) initiates a REPL for the Metacircular interpreter. You can exit it by click on EOF.

1. Study the code in lazymceval.rkt. Trace through what happens when the following expressions are evaluated. What would happen if this was run with the original (applicative order) interpreter was used?

   (define (try a b)
     (if (= a 0) 1 b))
   (try 0 (/ 1 0))

2. Study the code in lazymemmceval.rkt (memoized version of the lazy interpreter). Compare the execution of the following expressions in both the memoized and non-memoized versions of the lazy interpreter. What is the difference?

   (define (square x) (* x x))
   (define (sum-of-squares x y) (+ (square x) (square y)))
   (define (f a) (sum-of-squares (+ a 1) (* a 2)))
3. Study the code in ambeval.rkt (non-deterministic version of scheme).
   Trace through the execution of

   (define (prime-sum-pair list1 list2)
     (let ((a (an-element-of list1))
           (b (an-element-of list2)))
       (require (prime? (+ a b)))
       (list a b))
   (prime-sum-pair '(1 3 5 8) '(20 35 110))

   In order to run this example you must first enter the following definitions.

   (define (require p)
     (if (not p) (amb)))

   (define (an-element-of items)
     (require (not (null? items)))
     (amb (car items) (an-element-of (cdr items))))

   How many solutions are found [use try-again]? What are the solutions? First you might try

   (list (amb 1 2 3) (amb 'a 'b))

   To understand how amb works.

4. Modify the previous example to generate a list of prime numbers. Rather than searching
   through a list of potential primes, you will test integers greater than or equal to 2. Use the
   following function to generate such integers.

   (define (an-integer-starting-from n)
     (amb n (an-integer-starting-from (+ n 1))))