Metacircular Evaluator

1) In the meta-circular evaluator, if expressions are evaluated in the following way:
I. the predicate is evaluated
II. if the predicate is true, then the consequent expression is evaluated and returned
III. otherwise, the alternative expression is evaluated and returned

Consider the alternative definition of eval-if

```
(define (eval-if exp env)
  (let ((pred (mc-eval (if-predicate exp) env))
        (consequent (mc-eval (if-consequent exp) env))
        (alternative (mc-eval (if-alternative exp) env)))
    (if (true? pred)
        consequent
        alternative)))
```

a) Will this have the same semantics as if as we know it, meaning, does this definition of eval-if have the same behavior of the old eval-if?

b) Well, the answer is no, but explain why. If you can think of one situation where it breaks down, keep going. Think of three different kinds of scenarios where the behavior of this definition of eval-if is different, and write small programs to demonstrate your points.

2) Suppose we want to add a new special form called iff to the metacircular evaluator, that’s like an if where all the subexpressions after the predicate form the consequent, and there is no alternative. So while if can only take one subexpression as the consequent, and one subexpression as the alternative, iff can take any number of subexpressions as the consequent. If the predicate is true, all the subexpressions in the consequent should be evaluated in sequence, and the value of the iff is the value of the last subexpression. If the predicate is false, iff should produce false. For example:

```
> (iff (= 1 2) 3 4 5 6)
#f
> (iff (= 1 1) 3 4 5 6)
6
```

a. Is this mainly a change to the eval part or the apply part of the interpreter?

b. Make the appropriate changes to implement this feature.
3) When you write big scary programs, it’s helpful to know which parts of your big scary program the computer is spending most of its time running, so you know which parts to make more efficient. One simple way to do this is to count the number of times each procedure has been invoked.

Change our representation of procedures so that they keep track of how many times they have been invoked, and create a procedure that will return the number of times some procedure has been invoked. so I should be able to do stuff like this (counter returns the number of times the procedure has been invoked):

```scheme
(define (counter procedure)
  (procedure))
```

```scheme
(define (fact x)
  (if (= x 0)
      1
      (* x (fact (- x 1))))
)
```

```scheme
(counter fact)
0
(fact 10)
3628800
(counter fact)
11
```

Define counter as a special form (even though it doesn’t follow any special evaluation rules)

4) What’s the result of evaluating the following expressions using lexical and dynamic scope?

```scheme
(define x 5)
(define (foo x y)
  (bar y))
(define (bar y)
  (+ x y))
(foo 3 5)
```