Exercise 4.21.

Amazingly, Louis's intuition in exercise 4.20 is correct. It is indeed possible to specify recursive procedures without using `letrec` (or even `define`), although the method for accomplishing this is much more subtle than Louis imagined. The following expression computes `10` factorial by applying a recursive factorial procedure:[1]

```
((lambda (n)
   ((lambda (fact)
      (fact fact n))
    (lambda (ft k)
      (if (= k 1)
          1
          (* k (ft ft (- k 1)))))))
 10)
```

a. Check (by evaluating the expression) that this really does compute factorials. Devise an analogous expression for computing Fibonacci numbers.

b. Consider the following procedure, which includes mutually recursive internal definitions:

```
(define (f x)
  (define (even? n)
    (if (= n 0)
        true
        (odd? (- n 1))))
  (define (odd? n)
    (if (= n 0)
        false
        (even? (- n 1))))
  (even? x))
```

Fill in the missing expressions to complete an alternative definition of `f`, which uses neither internal definitions nor `letrec`:

```
(define (f x)
  ((lambda (even? odd?)
     (even? even? odd? x))
   (lambda (ev? od? n)
     (if (= n 0) true (od? <??> <??> <??>)))
   (lambda (ev? od? n)
     (if (= n 0) false (ev? <??> <??> <??>)))))
```

Answer.

a. We check the intention of this expression by exploiting the rule of substitution:

```
((lambda (fact)
   (fact fact 10))
 (lambda (ft k)
   (if (= k 1)
       1
       (* k (ft ft (- k 1))))))

((lambda (ft k)
   (if (= k 1)
```


Email address: informlarry@gmail.com

1. This example illustrates a programming trick for formulating recursive procedures without using `define`. The most general trick of this sort is the Y *operator*, which can be used to give a "pure λ-calculus" implementation of recursion. (See Stoy 1977 for details on the lambda calculus, and Gabriel 1988 for an exposition of the Y operator in Scheme.)

```
       1
       (* k (ft ft (- k 1)))))
 (lambda (ft k)
   (if (= k 1)
       1
       (* k (ft ft (- k 1)))))
 10)

(if (= 10 1)
    1
    (* 10
       ((lambda (ft k)
          (if (= k 1)
              1
              (* k (ft ft (- k 1)))))
        (lambda (ft k)
          (if (= k 1)
              1
              (* k (ft ft (- k 1)))))
        9)))

(* 10
   ((lambda (ft k)
          (if (= k 1)
              1
              (* k (ft ft (- k 1)))))
    (lambda (ft k)
          (if (= k 1)
              1
              (* k (ft ft (- k 1)))))
    9))

...

(* 10
   9
   8
   7
   6
   5
   4
   3
   2
   ((lambda (ft k)
          (if (= k 1)
              1
              (* k (ft ft (- k 1)))))
    (lambda (ft k)
          (if (= k 1)
              1
              (* k (ft ft (- k 1)))))
    1))

(* 10
   9
   8
   7
   6
   5
   4
```

```
   3
   2
   (if (= 1 1)
       1
       (* 1
          ((lambda (ft k)
             (if (= k 1)
                 1
                 (* k (ft ft (- k 1)))))
           (lambda (ft k)
             (if (= k 1)
                 1
                 (* k (ft ft (- k 1)))))
           0))))

(* 10 9 8 7 6 5 4 3 2 1)

3628800
```

The process generated shows that it really *does* compute factorials.

Besides, the computation process shows that the recursive factorial procedure is implemented by repeatly applying the inner most `lambda` expression to itself for a designated times. This pattern is essential and can be of great inspiration if we tries to devise an analogous recursive procedure. The following expression computes Fib (6) by applying a similar recursive Fibonacci procedure:

```
((lambda (n)
   ((lambda (fib)
      (fib fib n))
    (lambda (f k)
      (cond ((= k 0) 1)
            ((= k 1) 1)
            (else (+ (f f (- k 1))
                     (f f (- k 2))))))))
 6)
```

b.

```
(define (f x)
  ((lambda (even? odd?)
     (even? even? odd? x))
   (lambda (ev? od? n)
     (if (= n 0) true (od? ev? od? (- n 1))))
   (lambda (ev? od? n)
     (if (= n 0) false (ev? ev? od? (- n 1))))))
```