**Exercise 4.20.**

Because internal definitions look sequential but are actually simultaneous, some people prefer to avoid them entirely, and use the special form `letrec` instead. `Letrec` looks like `let`, so it is not surprising that the variables it binds are bound simultaneously and have the same scope as each other. The sample procedure `f` above can be written without internal definitions, but with exactly the same meaning, as

```
(define (f x)
  (letrec ((even?
            (lambda (n)
              (if (= n 0)
                  true
                  (odd? (- n 1)))))
           (odd?
            (lambda (n)
              (if (= n 0)
                  false
                  (even? (- n 1))))))
    <rest of body of f>))
```

`Letrec` expression, which have the form

```
(letrec ((<var_1> <exp_1>) ... (<var_n> <exp_n>))
  <body>)
```

are a variation on `let` in which the expressions <exp$_k$> that provide the initial values for the variables <var$_k$> are evaluated in an environment that includes all the `letrec` bindings. This permits recursion in the bindings, such as the mutual recursion of `even?` and `odd?` in the example above, or the evaluation of 10 factorial with

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

a. Implement `letrec` as a derived expression, by transforming a `letrec` expression into a `let` expression as shown in the text above or in exercise 4.18. That is, the `letrec` variables should be created with a `let` and then be assigned their values with `set!`.

b. Louis Reasoner is confused by all this fuss about internal definitions. The way he sees it, if you don't like to use `define` inside a procedure, you can just use `let`. Illustrate what is loose about his reasoning by drawing an environment diagram that shows the environment in which the `<rest of body of f>` is evaluated during evaluation of the expression `(f 5)`, with `f` defined as in this exercise. Draw an environment diagram for the same evaluation, but with `let` in place of `letrec` in the definition of `f`.

**Answer.**

a. To implement `letrec` as a derived expression, we include syntax procedures that extract parts of a `letrec` expression, and a procedure `letrec->let` that transforms `letrec` expressions into `let` expressions.

```
(define (letrec? exp) (tagged-list? exp 'letrec))

(define (letrec-bindings exp) (caddr exp))

(define (letrec-body exp) (cddr exp))
```

*. Creative Commons (CC BY-NC-SA) 2014, Lawrence X. Amlord (颜世敏, aka 颜 序).
Email address: informlarry@gmail.com

```scheme
(define (binding-variable binding) (car binding))

(define (letrec->let exp)
  (let ((bindings (letrec-bindings exp))
        (body (letrec-body exp)))
    (let ((vars (filter binding-variable bindings)))
      (let ((inits (map (lambda (var) (cons var '*unassigned*))
                        vars))
            (assignments (map (lambda (binding) (cons 'set! binding))
                              bindings)))
        (cons 'let
              (cons inits
                    (append assignments body)))))))
```

b. To draw out the environment diagram in evaluating `<rest body of f>` in `(f 5)`, we first substitute the `letrec` expression of `f` with an equivalent `let` expression.

```scheme
(define (f x)
  (let ((even? '*unassigned*)
        (odd? '*unassigned*))
    (set! even?
          (lambda (n)
            (if (= n 0)
                true
                (odd? (- n 1)))))
    (set! odd?
          (lambda (n)
            (if (= n 0)
                false
                (even? (- n 1)))))
    <rest of body of f>))
```

As shown in figure 1, we can see that both `even?` and `odd?` reside in the same environment, that is,

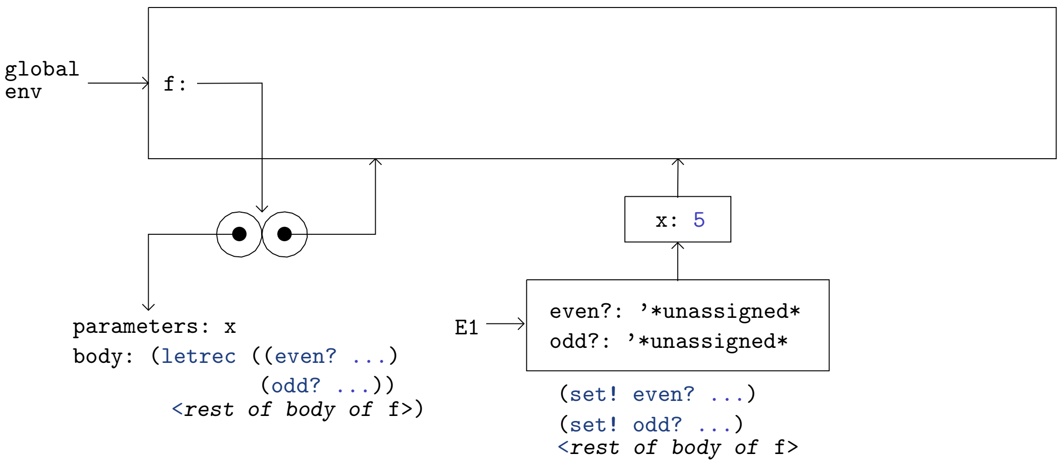


**Figure 1.** Environment structure produced by evaluating `<rest of body of f>` in `(f 5)` using `letrec`.

they have simultaneous scope. Therefore, mutual recursion can be dealt correctly and the evaluator would behave as what we originally expected.

However, following Louis's illusion, which simply use `let` in place of `letrec`, the behavior of the interpreter in evaluating `<rest of body of f>` varies a lot. As figure 2 shows, evaluating `even?` and `odd?` both create a procedure object whose environment are E2, rather than E1. This makes them invisible to each other when the interpreter comes to the evaluation of `<rest of body of f>`. Hence, by Louis's strategy, the evaluator will finally run into a sort of error like "`Unbound variable: even?`".

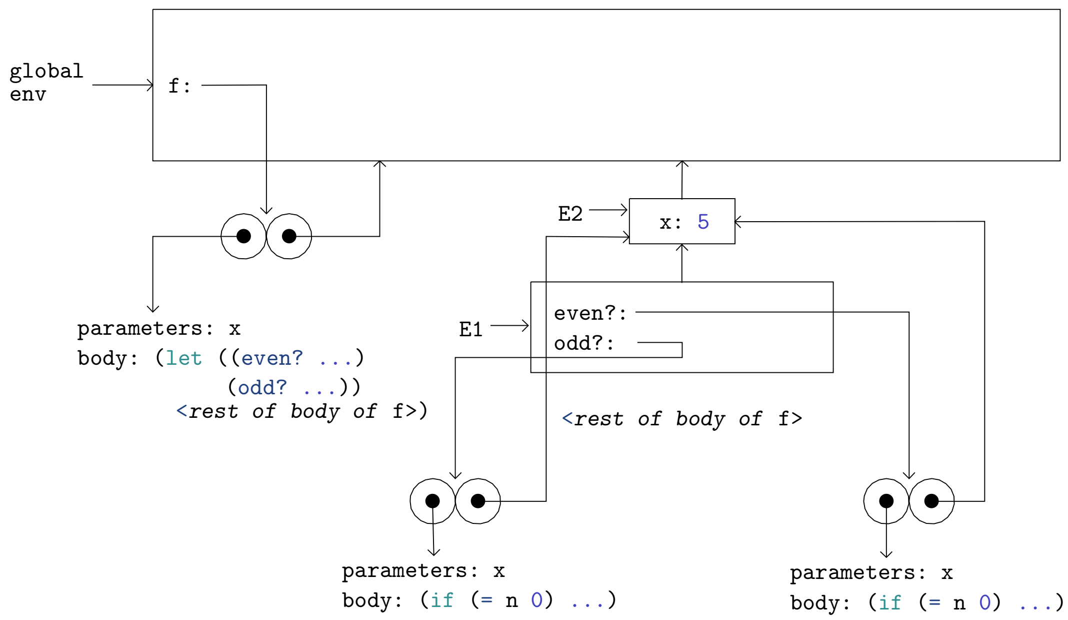


**Figure 2.** Environment structure produced by evaluating `<rest of body of f>` in `(f 5)` using `let`.