letrec
Here is a question you have seen before:
What does this evaluate to?

\[
\begin{align*}
&\text{(let ([f (lambda (x) (+ x 1))])} \\
&\quad \text{(let ([f (lambda (y) (if (= y 0) 10 (* 2 (f 0))))])}) \\
&\quad \text{(f 3))}
\end{align*}
\]

A. 2  
B. 4  
C. 10  
D. 20
Answer A: 2

(let ([f (lambda (x) (+ x 1))])
  (let ([f (lambda (y) (if (= y 0) 10 (* 2 (f 0))))])
    (f 3)))

The outer let makes an environment that binds f to "add 1"

In the inner let the lambda y expression is evaluated to a closure
whose closure environment has f bound to "add 1"
When we call (f 3) we evaluate the body of this closure in the closure
environment extended with a binding of y to 3. When we look up f in
this environment we get "add 1". So (* 2 (f 0)) evaluates to 2.
Why doesn't this work?

(let ([f (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))))] (f 5))

(let ([f (lambda (n) (if (= n 0) 1 (* n (f (- n 1)))))]) (f 5))
So what can we do to implement recursion?

We will have the parser parse a letrec expression such as

(letrec ([f exp1] [g exp2]) body)

into something equivalent that only involves things we have already implemented. We won't need to change eval-exp at all.
This will look stupid, but be patient.

What does this evaluate to?

(let ([f 0])
  (let ([g 34])
    (begin
      (set! f g)
      f))))
What does this evaluate to?

(let ([f 0])
  (let ([g (lambda (x) (+ 1 x))])
    (begin
      (set! f g)
      (f 5))))
What does this evaluate to?

(let ([f 0])
  (let ([g (lambda (x) (if (< 9 x) 10 (f (+ 1 x))))]))
    (begin
      (set! f g)
      (f 5))))

OK; so how do we write factorial with lets instead of letrec?
Answer:

(let ([fact 0])
  (let ([g (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))]))
    (begin
      (set! fact g)
      (fact 5))))
Here are some mutually recursive functions:

(letrec ([even? (lambda (x)
    (cond
      [(= 0 x) #t]
      [(= 1 x) #f]
      [else (odd? (- x 1)))]
    )]
[odd? (lambda (x)
    (cond
      [(= 0 x) #f]
      [(= 1 x) #t]
      [else (even? (- x 1)))]
    )])

(odd? 23)

How would you write this without letrec?
(let ([even? 0] [odd? 0])
  (let ([g1 (lambda (x)
              (cond
                [ (= 0 x) #t]
                [ (= 1 x) #f]
                [else (odd? (- x 1))]))])
   [g2 (lambda (x)
              (cond
                [ (= 0 x) #f]
                [ (= 1 x) #t]
                [else (even? (- x 1)))]))])
  (begin
    (set! even? g1)
    (set! odd? g2)
    (odd? 23)))
In general we want to replace

\[
\text{(letrec ([f}_1 \text{exp}_1] [f}_2 \text{exp}_2] \ldots [f}_n \text{exp}_n])
\text{body)}
\]

with

\[
\text{(let ([f}_1 0] [f}_2 0] \ldots [f}_n 0])
\text{(let ([g}_1 \text{exp}_1] [g}_2 \text{exp}_2] \ldots [g}_n \text{exp}_n])
\text{(begin}
\text{(set! f}_1 g}_1)
\text{(set! f}_2 g}_2)
\ldots
\text{(set! f}_n g}_n)
\text{body)))}
\]
How do we do that?
First, we need the g's to variables that don't appear anywhere else.

gensym is a Scheme function of no arguments that generates a new, unused symbol:

(gensym) might return a value such as 'g8035
Now, what are the pieces we have in an expression such as

\[
\text{input} = \quad (\text{letrec } ([f_1 \exp_1] [f_2 \exp_2] \ldots [f_n \exp_n])
\text{ body})
\]

We have

\[
\text{syms} = (f_1 \ldots f_n) = (\text{map car} \ (\text{cadr input}))
\]
\[
\text{exps} = (\exp_1 \ldots \exp_n) = (\text{map cadr} \ (\text{cadr input}))
\]
\[
\text{body} = (\text{caddr input})
\]
How do we build
(let ([f₁ 0] [f₂ 0] ... [fₙ 0]))

To build a let-exp for this we need (f₁...fₙ) We have that: syms

We need that many parsed 0s:
(map parse (map (lambda (x) 0) syms)) Isn't that clever???

We need the parsed body of this let expression. Its body is another let expression, which parses into another let-exp
The inner let is
   \[(\text{let } ([g_1 \ exp_1] [g_2 \ exp_2] \ ... \ [g_n \ exp_n]))\]
To build a let-exp for this we need
   \[
   \text{new-syms} = (g_1 \ ... \ g_n) \equiv (\text{map (lambda (x) (gensym)) syms})
   \]
   \[
   \text{parsed-exps} = (\text{map parse exps})
   \]
And the body of this is the begin expression
That begin expression is

\[(\text{begin} \ \ \ \\
\quad (\text{set! } f_1 \ g_1) \ \\
\quad (\text{set! } f_2 \ g_2) \ \\
\quad \ldots \ \\
\quad (\text{set! } f_n \ g_n) \ \\
\quad \text{body}))\]

You can generate the set!s with

\[(\text{map (lambda (x y) \ldots) syms new-syms})\]

and then append that onto \((\text{list (parse body)})\)
And then you are done and everything works!

You deserve to celebrate!!!