letrec

Here is a question you have seen before:

What does this evaluate to?

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

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))
```

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

$$
\begin{array}{l}
\texttt{(letrec ([}f_1\ exp_1\texttt{] [}f_2\ exp_2\texttt{] ... [}f_n\ exp_n\texttt{])} \\
\qquad\qquad \texttt{body)}
\end{array}
$$

with

$$
\begin{array}{l}
\texttt{(let ([}f_1\ 0\texttt{] [}f_2\ 0\texttt{] ... [}f_n\ 0\texttt{])} \\
\qquad\qquad \texttt{(let ([}g_1\ exp_1\texttt{] [}g_2\ exp_2\texttt{] ... [}g_n\ exp_n\texttt{])} \\
\qquad\qquad\qquad\qquad \texttt{(begin} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{(set! }f_1\ g_1\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{(set! }f_2\ g_2\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{...} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{(set! }f_n\ g_n\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{body)))}
\end{array}
$$

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

input = (letrec ([f_1 exp_1] [f_2 exp_2] ... [f_n exp_n])
body)

We have

syms = (f_1 ... f_n) = (map car (cadr input))
exps = (exp_1... exp_n) = (map cadr (cadr input))
body = (caddr input)

How do we build
(let ([f_1 0] [f_2 0] ... [f_n 0])

To build a let-exp for this we need (f_1...f_n) 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

$$\text{(let ([}g_1\ \text{exp}_1\text{] [}g_2\ \text{exp}_2\text{] ... [}g_n\ \text{exp}_n\text{])}$$

To build a let-exp for this we need

new-syms = (g1 ... gn) == (map (lambda (x) (gensym)) syms)

parsed-exps = (map parse exps)

And the body of this is the begin expression

That begin expression is

(begin
    (set! f_1 g_1)
    (set! f_2 g_2)
    ...
    (set! f_n g_n)
    body)))

You can generate the set!s with

(map (lambda (x y) ....) syms new-syms)

and then append that onto (list (parse body))

And then you are done and everything works!

You deserve to celebrate!!!