Factorial

```
(local [(define fac
          (lambda (n)
            (if (zero? n)
                1
                (* n (fac (- n 1))))))]
  (fac 10))
```

`local` binds both in the body expression and in the binding expression

Factorial

```
(let ([fac
       (lambda (n)
         (if (zero? n)
             1
             (* n (fac (- n 1)))))])
  (fac 10))
```

Doesn't work: **`let`** is like **`with`**

Still, at the point that we call **`fac`**, obviously we have a binding for **`fac`**...

... so pass it as an argument!

Factorial

```
(let ([facX
       (lambda (facX n)
         (if (zero? n)
             1
             (* n (facX facX (- n 1)))))])
  (facX facX 10))
```

Wrap this to get **fac** back...

Factorial

```
(let ([fac
       (lambda (n)
         (let ([facX
                (lambda (facX n)
                  (if (zero? n)
                      1
                      (* n (facX facX (- n 1)))))])
           (facX facX n)))])
  (fac 10))
```

Try this in the **HtDP Intermediate with Lambda** language, click **Step**

But the language we implement has only single-argument functions...

From Multi-Argument to Single-Argument

```
(define f
  (lambda (x y z)
    (list z y x)))

(f 1 2 3)
```

⇒

```
(define f
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (list z y x)))))

(((f 1) 2) 3)
```

Factorial

```
(let ([fac
       (lambda (n)
         (let ([facX
                (lambda (facX)
                  (lambda (n)
                    (if (zero? n)
                        1
                        (* n ((facX facX) (- n 1))))))])
           ((facX facX) n)))])
  (fac 10))
```

Simplify: `(lambda (n) (let ([f ...]) ((f f) n)))`
$\Rightarrow$ `(let ([f ...]) (f f))`...

Factorial

```
(let ([fac
       (let ([facX
              (lambda (facX)
                (lambda (n)
                   (if (zero? n)
                       1
                       (* n ((facX facX) (- n 1))))))])
         (facX facX))])
  (fac 10))
```

Factorial

```
(let ([fac
       (let ([facX
              (lambda (facX)
                ; Almost looks like original fac:
                (lambda (n)
                  (if (zero? n)
                      1
                      (* n ((facX facX) (- n 1))))))])
         (facX facX))])
  (fac 10))
```

More like original: introduce a local binding for `(facX facX)`...

Factorial

```
(let ([fac
       (let ([facX
              (lambda (facX)
                (let ([fac (facX facX)])
                  ; Exactly like original fac:
                  (lambda (n)
                    (if (zero? n)
                        1
                        (* n (fac (- n 1)))))))])
         (facX facX))])
  (fac 10))
```

Oops! – this is an infinite loop

We used to evaluate `(facX facX)` only when `n` is non-zero

Delay `(facX facX)`...

Factorial

```
(let ([fac
       (let ([facX
              (lambda (facX)
                (let ([fac (lambda (x)
                             ((facX facX) x))])
                  ; Exactly like original fac:
                  (lambda (n)
                    (if (zero? n)
                        1
                        (* n (fac (- n 1)))))))])
         (facX facX))])
  (fac 10))
```

Now, what about **`fib`**, **`sum`**, etc.?

Abstract over the **`fac`**-specific part...

Make-Recursive and Factorial

```
(define (mk-rec body-proc)
  (let ([fX
         (lambda (fX)
           (let ([f (lambda (x)
                      ((fX fX) x))])
             (body-proc f)))])
    (fX fX)))

(let ([fac (mk-rec
            (lambda (fac)
              ; Exactly like original fac:
              (lambda (n)
                (if (zero? n)
                    1
                    (* n (fac (- n 1)))))))])
  (fac 10))
```

Fibonnaci

```
(let ([fib
       (mk-rec
        (lambda (fib)
          ; Usual fib:
          (lambda (n)
            (if (or (= n 0) (= n 1))
                1
                (+ (fib (- n 1))
                   (fib (- n 2)))))))])
  (fib 5))
```

Sum

```
(let ([sum
       (mk-rec
        (lambda (sum)
          ; Usual sum:
          (lambda (l)
            (if (empty? l)
                0
                (+ (first l)
                   (sum (rest l)))))))])
  (sum '(1 2 3 4)))
```

Implementing Recursion

```
{rec {fac {fun {n}
               {ifzero n
                       1
                       {* n
                          {fac {- n 1}}}}}}}
 {fac 10}}
```

could be parsed the same as

```
{with {fac
       {mk-rec
        {fun {fac}
             {fun {n}
                  {ifzero n
                          1
                          {* n
                             {fac {- n 1}}}}}}}}
 {fac 10}}
```

Implementing Recursion

```
{rec {<id>1 <FAE>1}
  <FAE>2}
```

could be parsed the same as

```
{with {<id>1 {mk-rec {fun {<id>1} <FAE>1}}}
  <FAE>2}
```

which is really

```
{{fun {<id>1} <FAE>2}
 {mk-rec {fun {<id>1} <FAE>1}}}
```