Factorial

```scheme
(local [(define fac
          (λ (n)
            (if (zero? n)
                1
                (* n (fac (- n 1))))))]

(fac 10))
```
Factorial

(local [(define fac
    (λ (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
  (λ (n)
    (if (zero? n)
        1
        (* n (fac (- n 1)))))]
  (fac 10)))
Factorial

(let ([fac
  (λ (n)
   (if (zero? n)
     1
     (* n (fac (- n 1))))))]

(fac 10))

 Doesn’t work: let is like with
Factorial

(let ([fac
   (λ (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...
Factorial

(let ([fac
    (λ (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

  (λ (facX n)

    (if (zero? n)
        1
        (* n (facX facX (- n 1))))))]

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

Wrap this to get fac back...
Factorial

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

9
Factorial

(let ([fac
      (λ (n)
        (let ([facX
              (λ (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**
Factorial

(let ([fac
    (λ (n)
      (let ([facX
          (λ (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

(\texttt{define } f\texttt{ (x y z)}\newline \hspace{1cm} (\texttt{list z y x)))\newline \hspace{1cm} (f 1 2 3)\newline \Rightarrow\newline \hspace{1cm} (\texttt{define } f\texttt{ (x)}\newline \hspace{2cm} (\texttt{define } (y)\newline \hspace{3cm} (\texttt{define } (z)\newline \hspace{4cm} (\texttt{list z y x})))))\newline \hspace{1cm} (((f 1) 2) 3)
Factorial

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

13
Factorial

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

Simplify: (λ (n) (let ([f ...]) ((f f) n)))
    ⇒ (let ([f ...]) (f f))...
(let ([fac
  (let ([facX
    (λ (facX)
      (λ (n)
        (if (zero? n)
          1
          (* n ((facX facX) (- n 1))))))
    (facX facX))])
  (fac 10))
(let ([fac
    (let ([facX
        (λ (facX) ; Almost original fac:
            (λ (n)
                (if (zero? n)
                    1
                    (* n ((facX facX) (- n 1)))))
            (facX facX)))]
    (fac 10))
(let ([fac
   (let ([facX
                (λ (facX) ; Almost original fac:
                (λ (n)
                   (if (zero? n)
                    1
                    (* n ((facX facX) (- n 1)))))])
            (facX facX))]
      (fac 10))

More like original: introduce a local binding for
   (facX facX) ...
(let ([fac

  (let ([facX

    (λ (facX)

      (let ([fac (facX facX)])

        ; Exactly like original fac:

        (λ (n)

          (if (zero? n)

            1

            (* n (fac (- n 1)))))))]

      (facX facX)))]

    (fac 10))}
(let ([fac
  (let ([facX
    (let ([facX
      (λ (facX)
        (let ([fac (facX facX)])
          ; Exactly like original fac:
          (λ (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
Factorial

(let ([fac
      (let ([facX
            (λ (facX)
              (let ([fac (facX facX)])
                ; Exactly like original fac:
                (λ (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
    (λ (facX)
      (let ([fac (λ (x)
        ((facX facX) x)])]
        ; Exactly like original fac:
        (λ (n)
          (if (zero? n)
            1
            (* n (fac (- n 1)))))
        (facX facX)))]
    (fac 10)))]
    21)
Factorial

(let ([fac
    (let ([facX
        (let ([facX
            (let ([fac (let ([facX
                (let ([facX
                    (let ([fac (n)
                        (if (zero? n)
                            1
                            (* n (fac (- n 1))))))])
                    facX]
                facX)
            n)]]))]
        facX]
    fac)]))
(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
         (λ (fx)
           (let ([f (λ (x)
                       ((fx fx) x))]
                (body-proc f)))]
         (fx fx))]
  (let ([fac (mk-rec
              (λ (fac)
                ; Exactly like original fac:
                (λ (n)
                  (if (zero? n)
                      1
                      (* n (fac (- n 1))))))]
         (fac 10))

23
Fibonacci

(let ([fib
    (mk-rec
      (λ (fib)
        ;; Usual fib:
        (λ (n)
          (if (or (= n 0) (= n 1))
            1
            (+ (fib (- n 1))
              (fib (- n 2)))))))]
  (fib 5)))
(let ([sum
  (mk-rec
    (λ (sum)
      ; Usual sum:
      (λ (l)
        (if (empty? l)
          0
          (+ (first l)
            (sum (rest l))))))
    (sum '(1 2 3 4)))]

Sum