Part 1

Binding Constructs

```
{let {[x 5]}
  {+ x 6}}
```

```
{let {[f {lambda {x}
           {+ x 6}}]}
  {f 5}}
```

Encoding `let`

These programs are the same:

```
{let {[x 5]}
  {+ x 6}}
```

```
{{lambda {x}
   {+ x 6}}
 5}
```

Encoding `let`

These programs are the same:

```
{let {[x 5]}
  body}
```

```
{{lambda {x}
   body}
 5}
```

Encoding `let`

These programs are the same:

```
{let {[x rhs]}
  body}
```

```
{{lambda {x}
   body}
 rhs}
```

Encoding `let`

These programs are the same:

```
{let {[name rhs]}
  body}
```

```
{{lambda {name}
   body}
 rhs}
```

```
(test (parse '{let {[x 5]} {+ x 6}})
      (appC (lamC 'x (plusC (idC 'x) (numC 6)))
            (numC 5)))
```

Encoding Multiple Arguments

```
{let {[f {lambda {x y}
           {+ x y}}]}
  {f 1 2}}
```

```
{let {[f {lambda {x}
           {lambda {y}
             {+ x y}}}]}
  {{f 1} 2}}
```

Encoding Multiple Arguments

```
{let {[f {lambda {x y}
                body}]}
  {f 1 2}}
```

```
{let {[f {lambda {x}
                {lambda {y}
                  body}}]}
  {{f 1} 2}}
```

This transformation is called ***currying***

Part 2

Encoding `if`

```
{if tst
    thn
    els}
```

Encoding `if`

```
{if* tst
     {lambda {d} thn}
     {lambda {d} els}}
```

Encoding `if`

```
{{if* tst
      {lambda {d} thn}
      {lambda {d} els}}
 0}
```

```
true ≝ {lambda {x} {lambda {y} x}}
false ≝ {lambda {x} {lambda {y} y}}
```

```
{{{{tst
    {lambda {d} thn}}
   {lambda {d} els}}}
 0}
```

Encoding Pairs

```
{cons 1 empty}
```

Encoding Pairs

```
{pair 1 0}
```

Encoding Pairs

`{pair f r}`

Encoding Pairs

```
{lambda .... f r}
```

Encoding Pairs

```
{lambda {sel} {{sel f} r}}
```

```
pair ≝ {lambda {x}
         {lambda {y}
           {lambda {sel} {{sel x} y}}}}
fst ≝ {lambda {p} {p true}}
snd ≝ {lambda {p} {p false}}
```

```
{fst {{pair 1} 0}}
⇒ {fst {lambda {sel} {{sel 1} 0}}}
⇒ {{lambda {sel} {{sel 1} 0}} true}
⇒ {{true 1} 0}
=  {{{lambda {x} {lambda {y} x}} 1} 0}
⇒ {{lambda {y} 1} 0}
⇒ 1
```

Part 3

λ-Calculus Grammar

```
<Expr>  ::=  <Sym>
          |   {<Expr> <Expr>}
          |   {lambda {<Sym>} <Expr>}
```

λ-Calculus Grammar

```
<Expr>  ::=  <Sym>
          |  {<Expr> <Expr>}
          |  (λ (<Sym>) <Expr>)
```

```
true  ≝ (λ (x) (λ (y) x))
false ≝ (λ (x) (λ (y) y))
```

Part 4

Encoding Numbers

`zero` $\overset{\text{def}}{=}$ `(λ (x) (λ (y) y))`

Encoding Numbers

$$\texttt{zero} \overset{\text{def}}{=} \texttt{(}\lambda\texttt{ (f) (}\lambda\texttt{ (y) y))}$$

$$\texttt{one} \overset{\text{def}}{=} \texttt{(}\lambda\texttt{ (f) (}\lambda\texttt{ (y) \{f y\}))}$$

$$\texttt{two} \overset{\text{def}}{=} \texttt{(}\lambda\texttt{ (f) (}\lambda\texttt{ (y) \{f \{f y\}\}))}$$

$$\texttt{three} \overset{\text{def}}{=} \texttt{(}\lambda\texttt{ (f) (}\lambda\texttt{ (y) \{f \{f \{f y\}\}\}))}$$

$$N \overset{\text{def}}{=} \texttt{(}\lambda\texttt{ (f) (}\lambda\texttt{ (y) \{}\texttt{f}_1 \texttt{ ... \{}\texttt{f}_N\texttt{ y\}\}))}$$

Incrementing a Number

```
add1 ≝ (λ (n)
           ...)
```

Incrementing a Number

```
add1 ≝ (λ (n)
         (λ (f)
           (λ (x) ...)))
```

Incrementing a Number

```
add1 ≝ (λ (n)
         (λ (f)
           (λ (x) ... {{n f} x} ...)))
```

Incrementing a Number

```
add1 ≝ (λ (n)
         (λ (f)
           (λ (x) {f {{n f} x}}))))
```

```
(add1 zero)
⇒ (λ (f)
    (λ (x) {f {{zero f} x}}))
=  (λ (f)
    (λ (x) {f {{(λ (f) (λ (x) x)) f} x}}))
⇒ (λ (f)
    (λ (x) {f x}))
=  one
```

Adding Numbers

add2 $\overset{\text{def}}{=}$ (λ (n) {add1 {add1 n}})

add3 $\overset{\text{def}}{=}$ (λ (n) {add1 {add1 {add1 n}}})

add $\overset{\text{def}}{=}$ (λ (n) (λ (m) {add1$_1$ . . . {add1$_m$ n}}))

Adding Numbers

$\texttt{add2} \overset{\text{def}}{=}$ `(λ (n) {add1 {add1 n}})`

$\texttt{add3} \overset{\text{def}}{=}$ `(λ (n) {add1 {add1 {add1 n}}})`

$\texttt{add} \overset{\text{def}}{=}$ `(λ (n) (λ (m) {{m add1} n}))`

... because a number *m* applies some function *m* times to an argument

```
{{add one} two}
⇒ {{two add1} one}
⇒ {add1 {add1 one}}
⇒ three
```

Multiplying Numbers

$$
\begin{array}{l}
\texttt{mult} \overset{\text{def}}{=} (\lambda\ (\texttt{n})\ (\lambda\ (\texttt{m})\ \{\{\texttt{add n}\}_{1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{...} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \{\{\texttt{add n}\}_{\texttt{m}}\ \texttt{zero}\}\}))
\end{array}
$$

Multiplying Numbers

```
mult ≝ (λ (n) (λ (m) {{m {add n}} zero}))
```

... because `{add` *n*`}` is a function that adds *n* to any number

... and a number *m* applies some function *m* times to an argument

Testing for Zero

$$\texttt{iszero} \overset{\text{def}}{=} \texttt{(}\lambda\ \texttt{(n) ... true ... false ...)}$$

Testing for Zero

```
iszero ≝ (λ (n) {{n (λ (x) false)}
                  true})
```

because applying `(λ (x) false)` zero times to `true` produces `true`, and applying it any other number of times produces `false`

```
{iszero zero}
⇒ {{zero (λ (x) false)} true}
⇒ true
```

Testing for Zero

```
iszero ≝ (λ (n) {{n (λ (x) false)}
                    true})
```

because applying `(λ (x) false)` zero times to `true` produces `true`, and applying it any other number of times produces `false`

```
{iszero one}
⇒ {{one (λ (x) false)} true}
⇒ {(λ (x) false) true}
⇒ false
```

Decrementing a Number

```
sub1 ≝ (λ (n)
         (λ (f)
           (λ (x) ...)))
```

Decrementing a Number

```
sub1 ≝ (λ (n)
         (λ (f)
           (λ (x) ... {{n f} x} ...)))
```

Too late! No way to undo a call to `f`

Decrementing a Number

```
... {{pair zero} one}
... {{pair one} two}
... {{pair two} three}
...
... {{pair n-1} n}
```

Decrementing a Number

```
shift  ≝  (λ (p)
             {{pair {snd p}} {add1 {snd p}}})

sub1   ≝  (λ (n)
             {fst
              {{n shift} {{pair zero} zero}}})
```

And then subtraction is obvious...

Encodings

Using the minimal λ-calculus language we get

✓ functions

✓ local binding

✓ booleans

✓ numbers

Part 5

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

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

`letrec` hash the shape of **`let`** but the binding structure of **`local`**

Factorial

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

Doesn't work, because `let` binds `fac` only in the body

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 (fac (- n 1)))))])
  (facX facX 10))
```

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...

Part 6

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)))`

⇒ `(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
                (+ (fst l)
                   (sum (rest l)))))))])
  (sum '(1 2 3 4)))
```

Implementing Recursion

```
{letrec {[fac {lambda {n}
                {if0 n
                     1
                     {* n
                        {fac {- n 1}}}}}]}
  {fac 10}}
```

could be parsed the same as

```
{let {[fac
       {mk-rec
        {lambda {fac}
          {lambda {n}
            {if0 n
                 1
                 {* n
                    {fac {- n 1}}}}}}}]}
  {fac 10}}
```

Implementing Recursion

```
{letrec {[name rhs]}
  body}
```

could be parsed the same as

```
{let {[name {mk-rec {lambda {name} rhs}}]}
  body}
```

which is really

```
{{lambda {name} body}
 {mk-rec {lambda {name} rhs}}}
```

which, writing out *mk-rec*, is really

```
{{lambda {name} body}
 {{lambda {body-proc}
    {let {[fX {fun {fX}
                   {let {[f {lambda {x}
                              {{fX fX} x}}]}
                     {body-proc f}}}]}
      {fX fX}}}
  {lambda {name} rhs}}}
```