Part I

Binding Constructs

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

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

Converting `let` to `lambda`

These programs are the same:

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

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

Converting `let` to `lambda`

These programs are the same:

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

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

Converting `let` to `lambda`

These programs are the same:

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

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

Converting `let` to `lambda`

These programs are the same:

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

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

```
(test (parse `{let {[x 5]} {+ x 6}})
      (appE (lamE 'x (plusE (idE 'x) (numE 6)))
            (numE 5)))
```

Part 2

Syntactic Sugar and Libraries

We can add some features to Curly by changing only **parse**

```
(test (parse `{let {[x 5]} {+ x 6}})
      (appE (lamE 'x (plusE (idE 'x) (numE 6)))
            (numE 5)))
```

Language features that can be implemented this way are ***syntactic sugar***

Another example:

```
(test (parse `{neg 3})
      (multE (numE 3) (numE -1)))
```

... but that one might be better as just a function in a ***library***:

```
{let {[neg {lambda {n}
                   {* n -1}}]}
  ....}
```

Encodings

Syntactic sugar and library extensions are both are forms of ***encoding***

- Mutable variables encoded as boxes:

```
(test (parse '{lambda {x} {begin {set! x 1} x}})
      (lamE 'x (beginE (setboxE (idE 'x) (numE 1))
                       (unboxE (idE 'x)))))

(test (parse '{f 1})
      (appE (unboxE (idE 'f)) (boxE (numE 1))))
```

Encodings

Syntactic sugar and library extensions are both are forms of ***encoding***

- Boxes encoded with mutable variables:

```
{let {[crate
       {lambda {v}
         {lambda {sel}
           {{sel
             {lambda {x} v}}
            {lambda {x} {set! v x}}}}}]}
  {let {[uncrate
         {lambda {b}
           {{b {lambda {x} {lambda {y} x}}} 0}}]}
    {let {[set-crate!
           {lambda {b}
             {lambda {v}
               {{b {lambda {x} {lambda {y} y}}} v}}}]}
      ....}}}
```

Encodings

Syntactic sugar and library extensions are both are forms of ***encoding***

- Boxes encoded with mutable variables:

```
{let {[crate
       {lambda {v}
         {lambda {sel}
           {{sel
             {lambda {x} v}}
            {lambda {x} {set! v x}}}}}]}
  {let {[uncrate
         {lambda {b}
           {{b {lambda {x} {lambda {y} x}}} 0}}]}
    {let {[set-crate!
           {lambda {b}
             {lambda {v}
               {{b {lambda {x} {lambda {y} y}}} v}}}]}
      .... {{set-crate! b} 5} ....}}}
```

Syntactic Sugar in Libraries

Some languages, like Racket and Plait, support sugar via libraries

```
(define-syntax-rule (with [(v-id sto-id) call]
                          body)
  (type-case Result call
    [(v*s v-id sto-id) body]))


(define (interp [a : Exp] [env : Env] [sto : Store])
  (type-case Exp a
    ....
    [(plusE l r)
     (with [(v-l sto-l) (interp l env sto)]
       (with [(v-r sto-r) (interp r env sto-l)]
         (v*s (num+ v-l v-r) sto-r)))]
    ....))
```

Encodings and Expressiveness

Existing constructs determine what you can encode

- No state...

  ... no way to encode boxes or variables

- Just **`define`** for single-argument functions...

  ... no way to encode **`lambda`**

  ... no way to encode boxes

Encodings

Why study encodings:

- To identify language constructs that are fundamentally expressive

  e.g., boxes in contrast to `let`

- To simplify `interp`

  e.g., no `letE`

  ... but performance considerations may dominate

Part 3

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 4

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}
```

Part 5

Encoding Pairs

```
{cons 1 empty}
```

Encoding Pairs

```
{pair 1 0}
```

Encoding Pairs

```
{pair f s}
```

Encoding Pairs

```
{lambda .... f s}
```

Encoding Pairs

```
{lambda {sel} {{sel f} s}}
```

```
pair =def {lambda {x}
            {lambda {y}
              {lambda {sel} {{sel x} y}}}}
fst =def {lambda {p} {p true}}
snd =def {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 6

λ-Calculus Grammar

```
<Exp>  ::=  <Symbol>
         |  {<Exp> <Exp>}
         |  {lambda {<Symbol>} <Exp>}
```

λ-Calculus Grammar

```
<Exp>  ::=  <Symbol>
        |   {<Exp> <Exp>}
        |   (λ (<Symbol>) <Exp>)
```

$$\texttt{true} \stackrel{\text{def}}{=} \texttt{(λ (x) (λ (y) x))}$$

$$\texttt{false} \stackrel{\text{def}}{=} \texttt{(λ (x) (λ (y) y))}$$

λ-Calculus Grammar

```
<Exp>  ::=  <Symbol>
         |  {<Exp> <Exp>}
         |  (λ (<Symbol>) <Exp>)
```

```
{{true a} b}
```

λ-Calculus Grammar

```
<Exp>  ::=  <Symbol>
         |  {<Exp> <Exp>}
         |  (λ (<Symbol>) <Exp>)
```

```
{{(λ (x) (λ (y) x)) a} b}
```

Part 7

Encoding Numbers

$$\texttt{zero} \stackrel{\text{def}}{=} \texttt{(}\lambda\ \texttt{(x)}\ \ \texttt{(}\lambda\ \texttt{(y)}\ \texttt{y))}$$

Encoding Numbers

`zero` $\stackrel{\text{def}}{=}$ `(λ (f) (λ (x) x))` applies `f` to `x` zero times

`one` $\stackrel{\text{def}}{=}$ `(λ (f) (λ (x) {f x}))` applies `f` to `x` one time

`two` $\stackrel{\text{def}}{=}$ `(λ (f) (λ (x) {f {f x}}))`

`three` $\stackrel{\text{def}}{=}$ `(λ (f) (λ (x) {f {f {f x}}}))`

N $\stackrel{\text{def}}{=}$ `(λ (f) (λ (x) {`f_1 `. . . {`f_N `x}}))` `f` to `x` ***N*** times

This encoding is called ***Church numerals***

Incrementing a Number

```
add1 =def (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
```

Part 8

Adding Numbers

$$\texttt{add2} \stackrel{\text{def}}{=} \texttt{(}\lambda\ \texttt{(n)}\ \texttt{\{add1 \{add1 n\}\})}$$

$$\texttt{add3} \stackrel{\text{def}}{=} \texttt{(}\lambda\ \texttt{(n)}\ \texttt{\{add1 \{add1 \{add1 n\}\}\})}$$

$$\texttt{add} \stackrel{\text{def}}{=} \texttt{(}\lambda\ \texttt{(n)}\ \texttt{(}\lambda\ \texttt{(m)}\ \texttt{\{add1}_{\texttt{|}}\ \texttt{...}\ \texttt{\{add1}_{\texttt{m}}\ \texttt{n\}\}))}$$

Adding Numbers

```
add2 ≝ (λ (n) {add1 {add1 n}})

add3 ≝ (λ (n) {add1 {add1 {add1 n}}})

add ≝ (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

```
mult =def (λ (n) (λ (m) {{add n}1
                         ...
                         {{add n}m zero}}))
```

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

```
iszero ≝ (λ (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} zero}
... {{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}}})
```

```
{shift {{pair zero} zero}} ⇒ {{pair zero} one}
{shift {{pair zero} one}}  ⇒ {{pair one} two}
{shift {{pair n-2} n-1}}   ⇒ {{pair n-1} n}
```

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

And then subtraction is obvious...

Part 9

More Numbers

Negative integers: pair non-negative integer with a sign boolean

Rational numbers: pair numerator and denominator

Complex numbers: pair real and imaginary parts

Encodings

Using the minimal λ-calculus language we get

- ✓ functions
- ✓ local binding
- ✓ booleans
- ✓ numbers