Part I

Binding Constructs

```
let x = 5:
  x + 6


let f = (fun (x):
             x + 6):
  f(5)
```

Converting `let` to `lambda`

These programs are the same:

```
let x = 5:
  x + 6


(fun (x):
   x + 6)(5)
```

Converting `let` to `lambda`

These programs are the same:

```
let x = 5:
  x + 6


(fun (x): x + 6)(5)
```

Converting `let` to `lambda`

These programs are the same:

```
let x = 5:
  body



(fun (x): body)(5)
```

Converting `let` to `lambda`

These programs are the same:

```
let x = rhs:
  body


(fun (x): body)(rhs)
```

Converting `let` to `lambda`

These programs are the same:

```
let name = rhs:
  body


(fun (name): body)(rhs)
```

```
check: parse('let x = 5: x + 6')
         ~is appE(funE(#'x, plusE(idE(#'x), intE(6))),
                  intE(5))
```

Part 2

Syntactic Sugar and Libraries

We can add some features to Moe by changing only **`parse`**

```
check: parse('let x = 5: x + 6')
       ~is appE(funE(#'x, plusE(idE(#'x), intE(6))),
                intE(5))
```

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

Another example:

```
check: parse('- 3')
       ~is multE(intE(3), intE(-1))
```

Syntactic Sugar and Libraries

We can add some features to Moe by changing only `parse`

```
check: parse('let x = 5: x + 6')
       ~is appE(funE(#'x, plusE(idE(#'x), intE(6))),
                intE(5))
```

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

Another example:

```
check: parse('neg(3)')
       ~is multE(intE(3), intE(-1))
```

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

```
let neg = (fun (n):
                 n * -1):
  ....
```

Encodings

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

- Mutable variables encoded as boxes:

```
check: parse('fun (x): begin: x := 1; x')
       ~is funE(#'x, beginE(setboxE(idE(#'x), intE(1)),
                            unboxE(idE(#'x))))

check: parse('f(1)')
       ~is appE(unboxE(idE(#'f)), boxE(intE(1)))
```

Encodings

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

- Boxes encoded with mutable variables:

```
let crate = (fun (v):
               fun (sel):
                 sel(fun (x): v)(fun (x): v := x)):
  let uncrate = (fun (b):
                   b(fun (x): fun (y): x)(0)):
    let set_crate = (fun (b):
                       fun (v):
                         b(fun (x): fun (y): y)(v)):
      ....
```

Encodings

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

- Boxes encoded with mutable variables:

```
let crate = (fun (v):
               fun (sel):
                 sel(fun (x): v)(fun (x): v := x)):
  let uncrate = (fun (b):
                   b(fun (x): fun (y): x)(0)):
    let set_crate = (fun (b):
                       fun (v):
                         b(fun (x): fun (y): y)(v)):
      .... set_crate(b)(5) ....
```

Syntactic Sugar in Libraries

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

```
macro 'reslet ($v_id, $sto_id) = $call:
           $body':
  'match $call
   | res($v_id, $sto_id): $body'


fun interp(a :: Exp, env :: Env, sto :: Store):
  match a
  | ....
  | plusE(l, r):
      reslet (v_l, sto_l) = interp(l, env, sto):
        reslet (v_r, sto_r) = interp(r, env, sto_l):
          res(num_plus(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 definition-style **`fun`** for single-argument functions...

... no way to encode lambda-style **`fun`**

... 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 = (fun (x, y):
                x + y):
  f(1, 2)


let f = (fun (x):
                fun (y):
                  x + y):
  f(1)(2)
```

Encoding Multiple Arguments

```
let f = (fun (x, y):
             body):
  f(1, 2)


let f = (fun (x):
             fun (y):
               body):
  f(1)(2)
```

This transformation is called ***currying***

Part 4

Encoding `if`

```
if tst
| thn
| els
```

Encoding `if`

```
if_proc(tst,
        fun (d): thn,
        fun (d): els)
```

Encoding `if`

```
if_proc(tst,
        fun (d): thn,
        fun (d): els)(0)
```

`true` $\stackrel{\text{def}}{=}$ `fun (x): fun (y): x`

`false` $\stackrel{\text{def}}{=}$ `fun (x): fun (y): y`

```
tst(fun (d): thn)(fun (d): els)(0)
```

Part 5

Encoding Pairs

```
cons(1, empty)
```

Encoding Pairs

```
pair(1, 0)
```

Encoding Pairs

```
pair(f, s)
```

Encoding Pairs

```
fun .... f s
```

Encoding Pairs

```
fun (sel): sel(f)(s)
```

pair $\stackrel{\text{def}}{=}$
```
fun (x):
  fun (y):
    fun (sel): sel(x)(y)
```
fst $\stackrel{\text{def}}{=}$ `fun (p): p(true)`

snd $\stackrel{\text{def}}{=}$ `fun (p): p(false)`

```
fst(pair(1)(0))
```
$\Rightarrow$ `fst(fun (sel): sel(1)(0))`

$\Rightarrow$ `(fun (sel): sel(1)(0))(true)`

$\Rightarrow$ `true(1)(0)`

= `(fun (x): fun (y): x)(1)(0)`

$\Rightarrow$ `(fun (y): 1)(0)`

$\Rightarrow$ `1`

Part 6

λ-Calculus Grammar

```
<Exp>  ::=  <Symbol>
        |   <Exp>(<Exp>)
        |   fun (<Symbol>): <Exp>
```

λ-Calculus Grammar

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

$$\texttt{true} \stackrel{\text{def}}{=} \lambda(x):\ \lambda(y):\ x$$
$$\texttt{false} \stackrel{\text{def}}{=} \lambda(x):\ \lambda(y):\ y$$

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

```
zero ≝ λ(x): λ(y): 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 Int

```
add1 ≝ λ(n):
             ....
```

Incrementing a Int

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

Incrementing a Int

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

Incrementing a Int

```
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

```
add2 ≝ λ(n): add1(add1(n))

add3 ≝ λ(n): add1(add1(add1(n)))

add ≝ λ(n): λ(m): add1₁(...(add1ₘ(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

$$\texttt{mult} \stackrel{\text{def}}{=} \lambda\texttt{(n): } \lambda\texttt{(m): add(n)}_{1}\texttt{(...(add(n)}_{m}\texttt{(zero)))}$$

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

$$\texttt{iszero} \stackrel{\text{def}}{=} \lambda\texttt{(n): n(}\lambda\texttt{(x): false)(true)}$$

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

$$\texttt{iszero} \stackrel{\text{def}}{=} \texttt{λ(n): n(λ(x): false)(true)}$$

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 Int

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

Decrementing a Int

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

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

Decrementing a Int

```
.... pair(zero)(zero)
.... pair(zero)(one)
.... pair(one)(two)
.... pair(two)(three)
....
.... pair(n-1)(n)
```

Decrementing a Int

```
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