Part I

Shrinking the Language

Shrinking the Language

- We've seen that **`with`** is not really necessary when we have **`fun`**...
- ... and **`rec`** is not really necessary when we have **`fun`**...
- ... and neither, it turns out, are fancy things like numbers, **`+`**, **`-`** or **`if0`**

This part's material won't show up on any homework or exam

LC Grammar

```
<LC> ::= <id>
      |  {<LC> <LC>}
      |  {fun {<id>} <LC>}
```

Implementing Programs with LC

Can you write a program that produces the identity function?

```
{fun {x} x}
```

Implementing Programs with LC

Can you write a program that produces zero?

What's *zero*? I only know how to write functions!

Turing Machine programmer: What's a *function*? I only know how to write 0 or 1!

We need to encode zero — instead of agreeing to write zero as **0**, let's agree to write it as

```
{fun {f} {fun {x} x}}
```

This encoding is the start of ***Church numerals***...

Implementing Numbers with LC

Can you write a program that produces zero?

```
{fun {f} {fun {x} x}}
```

... which is also the function that takes `f` and `x` and applies `f` to `x` zero times

From now on, we'll write `zero` as shorthand for the above expression:

$$\texttt{zero} \overset{\text{def}}{=} \texttt{\{fun \{f\} \{fun \{x\} x\}\}}$$

Implementing Numbers with LC

Can you write a program that produces one?

$$\texttt{one} \stackrel{\text{def}}{=} \texttt{\{fun \{f\} \{fun \{x\} \{f x\}\}\}}$$

... which is also the function that takes **`f`** and **`x`** and applies **`f`** to **`x`** one time

Implementing Numbers with LC

Can you write a program that produces two?

$$\texttt{two} \stackrel{\text{def}}{=} \texttt{\{fun \{f\} \{fun \{x\} \{f \{f x\}\}\}\}}$$

... which is also the function that takes **`f`** and **`x`** and applies **`f`** to **`x`** two times

Implementing Booleans with LC

Can you write a program that produces true?

$$\texttt{true} \overset{\text{def}}{=} \texttt{\{fun \{x\} \{fun \{y\} x\}\}}$$

... which is also the function that takes two arguments and returns the first one

Implementing Booleans with LC

Can you write a program that produces false?

```
false ≝ {fun {x} {fun {y} y}}
```

... which is also the function that takes two arguments and returns the second one

Implementing Branches with LC

```
true   ≝ {fun {x} {fun {y} x}}
false  ≝ {fun {x} {fun {y} y}}
zero   ≝ {fun {f} {fun {x} x}}
one    ≝ {fun {f} {fun {x} {f x}}}
two    ≝ {fun {f} {fun {x} {f {f x}}}}
```

Can you write a program that produces zero when given true, one when given false?

```
{fun {b} {{b zero} one}}
```

... because `true` returns its first argument and `false` returns its second argument

```
{{fun {b} {{b zero} one}} true} ⇒ {{true zero} one}
                                ⇒ zero

{{fun {b} {{b zero} one}} false} ⇒ {{false zero} one}
                                 ⇒ one
```

Implementing Pairs

Can you write a program that takes two arguments and produces a pair?

```
cons ≝ {fun {x} {fun {y}
                    {fun {b} {{b x} y}}}}
```

Examples:

```
{{cons zero} one} ⇒ {fun {b} {{b zero} one}}

{{cons two} zero} ⇒ {fun {b} {{b two} zero}}
```

Implementing Pairs

```
cons ≝ {fun {x} {fun {y}
                  {fun {b} {{b x} y}}}}
```

Can you write a program that takes a pair and returns the first part?

Can you write a program that takes a pair and returns the rest?

```
first ≝ {fun {p} {p true}}

rest ≝ {fun {p} {p false}}
```

Example:

```
{first {{cons zero} one}} ⇒ {first {fun {b} {{b zero} one}}}
                          ⇒ {{fun {b} {{b zero} one}} true}
                          ⇒ {{true zero} one}
                          ⇒ zero
```

Implementing Arithmetic

```
zero  ≝  {fun {f} {fun {x} x}}

one   ≝  {fun {f} {fun {x} {f x}}}

two   ≝  {fun {f} {fun {x} {f {f x}}}}
```

Can you write a program that takes a number and adds one?

```
add1 ≝ {fun {n}
         {fun {g} {fun {y}
                    {g {{n g} y}}}}}
```

Example:

```
{add1 zero} ⇒ {fun {g} {fun {y}
                          {g {{zero g} y}}}}
            = {fun {g} {fun {y}
                          {g {{{fun {f} {fun {x} x}} g} y}}}}
            ⇔ {fun {g} {fun {y}
                          {g y}}}
            = one
```

Implementing Arithmetic

Can you write a program that takes a number and adds two?

$$\texttt{add2} \overset{\text{def}}{=} \texttt{\{fun \{n\} \{add1 \{add1 n\}\}\}}$$

Implementing Arithmetic

Can you write a program that takes a number and adds three?

$\texttt{add3} \stackrel{\text{def}}{=}$ `{fun {n} {add1 {add1 {add1 n}}}}`

Implementing Arithmetic

```
zero ≝ {fun {f} {fun {x} x}}
one  ≝ {fun {f} {fun {x} {f x}}}
two  ≝ {fun {f} {fun {x} {f {f x}}}}
```

Can you write a program that takes two numbers and adds them?

```
add ≝ {fun {n} {fun {m} {{n add1} m}}}
```

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

Implementing Arithmetic

```
zero ≝ {fun {f} {fun {x} x}}
one  ≝ {fun {f} {fun {x} {f x}}}
two  ≝ {fun {f} {fun {x} {f {f x}}}}
```

Can you write a program that takes two numbers and multiplies them?

```
mult ≝ {fun {n} {fun {m} {{n {add m}} zero}}}
```

... because adding number *m* to zero *n* times produces $n \times m$

Implementing Arithmetic

Can you write a program that tests for zero?

`iszero` $\stackrel{\text{def}}{=}$ `{fun {n} {{n {fun {x} false}} true}}`

because applying `{fun {x} false}` zero times to `true` produces `true`, and applying it any other number of times produces `false`

Implementing Arithmetic

Can you write a program that takes a number and produces one less?

```
shift  ≝  {fun {p}
               {{cons {rest p}} {add1 {rest p}}}}

sub1   ≝  {fun {n}
               {first
                {{n shift} {{cons zero} zero}}}}
```

And then subtraction is obvious...

Implementing Factorial

```
mk-rec ≝ {fun {body}
              {{fun {fX} {fX fX}}
               {fun {fX}
                    {{fun {f} {body f}}
                     {fun {x} {{fX fX} x}}}}}}
```

Can you write a program that computes factorial?

```
{mk-rec
 {fun {fac}
      {fun {n}
           {{{iszero n}
             one}
            {{mult n} {fac {sub1 n}}}}}}}
```

... and when you can write factorial, you can probably write anything.

Part II

Back to Recursive Binding

Recursive Binding

```
{rec {x x} x}
```

infinite loop

Recursive Binding

```
{with {f {fun {g} {g g}}}
  {f f}}
```

infinite loop

Recursive Binding

```
(local [(define x x)]
  x)
```

```
#<undefined>
```

Recursive Binding

```
{rec {x x} 10}
```

infinite loop

Recursive Binding

```
(local [(define x x)]
  10)
```

```
10
```

Recursive Binding

```
(local [(define x 10)]
  x)
```

10

Recursive Binding

```
(local [(define x (list x))]
  x)
```

```
(list #<undefined>)
```

Recursive Binding

```
(local [(define (f x) (f x))]
  (f 1))
```

infinite loop

Recursive Binding

```
(local [(define f
          (lambda (x) (f x)))]
  (f 1))
```

infinite loop

Recursive Binding

```
(local [(define f
          (list
           (lambda (x) ((first f) x))))]
  ((first f) 1))
```

infinite loop

Recursive Binding

```
(local [(define val
          (interp (num 10)
                  (aSub 'x
                        val
                        ds)))]
  val)
```

contract failure

Recursive Binding

```
(local [(define val
          (interp (num 10)
                  (aSub 'x
                        (lambda () val)
                       ds)))]
  val)
```

could work

Recursive Binding

```
(local [(define new-ds
          (aSub 'x
                (lambda () val)
                ds))
        (define val
          (interp (num 10)
                  new-ds))]
  (interp (id 'x) new-ds))
```

could work

Metacircular Recursion

```
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?)
        (get-value (-> FAE-Value?))
        (rest DefrdSub?)])

(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free variable")]
    [aSub (sub-name get-num rest-ds)
          (if (symbol=? sub-name name)
              (get-num)
              (lookup name rest-ds))]))
```

Metacircular Recursion

```
(define-type FAE
  ....
  [rec (name symbol?)
       (named-expr FAE?)
       (body FAE?)])

(define (interp a-fae ds)
  (type-case FAE a-fae
    ....
    [rec (name named-expr body-expr)
         (local [(define new-ds
                   (aSub name
                         (lambda () val)
                         ds))
                 (define val (interp named-expr
                                     new-ds))]
           (interp body-expr new-ds))]))
```