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

\[
\text{<LC> ::= <id>}
\]

\[
| \{<LC> <LC>\}
\]

\[
| \{\text{fun} \{<id>\} <LC>\}
\]
Implementing Programs with LC

Can you write a program that produces the identity function?

\{\textbf{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

   \{\text{fun } \{f\} \ {\text{fun } \{x\} \ x}\}\}

This encoding is the start of Church numerals...
Implementing Numbers with LC

Can you write a program that produces zero?

\[ \text{fun \{f\} \text{fun \{x\} x}} \]  

... which is also the function that takes \( f \) and \( x \) and applies \( f \) to \( x \) zero times

From now on, we’ll write \text{zero} as shorthand for the above expression:

\[ \text{zero} \overset{\text{def}}{=} \text{fun \{f\} \text{fun \{x\} x}} \]
Implementing Numbers with LC

Can you write a program that produces one?

\[
\text{one} \overset{\text{def}}{=} \{\text{fun } \{f\} \{\text{fun } \{x\} \{f \ x\}\}\}\}
\]

... 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?

\[
two = \{ \text{fun } f \} \{ \text{fun } x \} \{ f \{ f \ x \}\}\}
\]

... 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?

\[
\text{true} \overset{\text{def}}{=} \{\text{fun} \ {x} \ \{\text{fun} \ {y} \ x\}\}
\]

... 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?

\[
\text{false} \overset{\text{def}}{=} \{\text{fun } \{x\} \{\text{fun } \{y\} y\}\}
\]

... which is also the function that takes two arguments and returns the second one.
Implementing Branches with LC

\[
\begin{align*}
true & \overset{\text{def}}{=} \{\text{fun } \{x\} \{\text{fun } \{y\} x\}\} \\
false & \overset{\text{def}}{=} \{\text{fun } \{x\} \{\text{fun } \{y\} y\}\} \\
zero & \overset{\text{def}}{=} \{\text{fun } \{f\} \{\text{fun } \{x\} x\}\} \\
one & \overset{\text{def}}{=} \{\text{fun } \{f\} \{\text{fun } \{x\} \{f\ \{f\ \{x\}\}\}\}\} \\
two & \overset{\text{def}}{=} \{\text{fun } \{f\} \{\text{fun } \{x\} \{f\ \{f\ \{x\}\}\}\}\}\
\end{align*}
\]

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

\[
\{\text{fun } \{b\} \{\{b\ \text{zero}\} \text{ one}\}\}\]
\]

... because \textit{true} returns its first argument and \textit{false} returns its second argument

\[
\begin{align*}
\{\{\text{fun } \{b\} \{\{b\ \text{zero}\} \text{ one}\}\} \text{ true}\} & \Rightarrow \{\{\text{true}\ \text{zero}\} \text{ one}\} \\
& \Rightarrow \text{zero} \\
\{\{\text{fun } \{b\} \{\{b\ \text{zero}\} \text{ one}\}\} \text{ false}\} & \Rightarrow \{\{\text{false}\ \text{zero}\} \text{ one}\} \\
& \Rightarrow \text{one}
\end{align*}
\]
Implementing Pairs

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

\[
\text{cons} \overset{\text{def}}{=} \{\text{fun } x\} \{\text{fun } y\} \{\text{fun } b\} \{\{b \ x\ y\}\}\}
\]

Examples:

\[
\{\{\text{cons zero} \ \text{one}\}\} \Rightarrow \{\text{fun } b\} \{\{b \ \text{zero} \ \text{one}\}\}\}
\]

\[
\{\{\text{cons two} \ \text{zero}\}\} \Rightarrow \{\text{fun } b\} \{\{b \ \text{two} \ \text{zero}\}\}\}
\]
Implementing Pairs

\[
\text{cons} \overset{\text{def}}{=} \{\text{fun } x \{\text{fun } y \{\text{fun } b \{\{b \ x \} \ y\}\}\}\}\}
\]

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?

\[
\text{first} \overset{\text{def}}{=} \{\text{fun } p \{p \ \text{true}\}\}
\]

\[
\text{rest} \overset{\text{def}}{=} \{\text{fun } p \{p \ \text{false}\}\}
\]

Example:

\[
\{\text{first} \{\{\text{cons zero} \ \text{one}\}\}\} \Rightarrow \{\text{first} \{\text{fun } b \{\{b \ \text{zero} \} \ \text{one}\}\}\}\}
\Rightarrow \{\{\text{fun } b \{\{b \ \text{zero} \} \ \text{one}\}\} \ \text{true}\}
\Rightarrow \{\{\text{true zero} \ \text{one}\}\}
\Rightarrow \text{zero}
\]
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?

\[
\text{add2} \overset{\text{def}}{=} \{\text{fun } n\{\text{add1 } \{\text{add1 } n\}\}\}\]

Implementing Arithmetic

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

\[
\text{add3} \overset{\text{def}}{=} \{ \text{fun} \{n\} \{ \text{add1} \{ \text{add1} \{ \text{add1} \ n \} \} \} \} \]

Implementing Arithmetic

\[
\begin{align*}
\text{zero} & \overset{\text{def}}{=} \{\text{fun} \ f \ \{\text{fun} \ x \ x\}\} \\
\text{one} & \overset{\text{def}}{=} \{\text{fun} \ f \ \{\text{fun} \ x \ f \ x\}\} \\
\text{two} & \overset{\text{def}}{=} \{\text{fun} \ f \ \{\text{fun} \ x \ f \ f \ x\}\}\}
\end{align*}
\]

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

\[
\text{add} \overset{\text{def}}{=} \{\text{fun} \ n \ \{\text{fun} \ m \ \{\{n \ \text{add1} \} \ m\}\}\}
\]

... because a number \( n \) applies some function \( n \) times to an argument
Implementing Arithmetic

\[
\begin{align*}
\text{zero} & \overset{\text{def}}{=} \{\text{fun} \ (f) \ \{\text{fun} \ (x) \ x\}\} \\
\text{one} & \overset{\text{def}}{=} \{\text{fun} \ (f) \ \{\text{fun} \ (x) \ (f \ x)\}\} \\
\text{two} & \overset{\text{def}}{=} \{\text{fun} \ (f) \ \{\text{fun} \ (x) \ (f \ (f \ x))\}\}\}
\end{align*}
\]

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

\[
\text{mult} \overset{\text{def}}{=} \{\text{fun} \ (n) \ \{\text{fun} \ (m) \ \{\{n \ \{\text{add} \ m\}\} \ \text{zero}\}\}\}
\]

... because adding number \( m \) to zero \( n \) times produces \( n \times m \)
Implementing Arithmetic

Can you write a program that tests for zero?

```
iszzero = (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?

\[
\text{shift} \quad \text{def} \quad \{ \text{fun} \quad \{ p \} \\
\quad \quad \quad \{ \{ \text{cons} \quad \{ \text{rest} \quad p \} \} \quad \{ \text{add1} \quad \{ \text{rest} \quad p \} \} \} \}\]

\[
\text{subl} \quad \text{def} \quad \{ \text{fun} \quad \{ n \} \\
\quad \quad \quad \{ \text{first} \\
\quad \quad \quad \quad \{ \{ n \quad \text{shift} \} \quad \{ \{ \text{cons} \quad \text{zero} \} \quad \text{zero} \} \} \} \}
\]

And then subtraction is obvious...
Implementing Factorial

```markdown
\[ \text{mk-rec} \equiv \{ \text{fun} \ \{ \text{body} \} \{
\{\text{fun} \ \{\text{fx}\} \ \{\text{fx} \ \text{fx}\}\}
\{\text{fun} \ \{\text{fx}\} \{
\{\text{fun} \ \{f\} \ \{\text{body} \ f\}\}
\{\text{fun} \ \{x\} \ \{\{\text{fx} \ \text{fx}\} \ \{\text{x}\}\}\}\}\}\}\}\}\]
```

Can you write a program that computes factorial?

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

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

Back to Recursive Binding
Recursive Binding

\{\texttt{rec} \{x \ x\} \ x\}

infinite loop
Recursive Binding

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

infinite loop
Recursive Binding

(llocal [(define x x)]
  x)

#<undefined>
Recursive Binding

{rec {x x} 10}

infinite loop
Recursive Binding

\[
\text{(local [(define x x)]
\text{ 10})}
\]

\text{10}
Recursive Binding

(llocal [(define x 10)]
  x)

10
Recursive Binding

\[
\text{(local [\((\text{define } x (\text{list } x))\)] } \ x) \\
\text{(list } \#<\text{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

\[
\text{(local } [(\text{define } \text{val} \\\n\text{  (interp (num 10) } \\\n\text{    (aSub 'x } \\\n\text{      (\lambda () \text{val}) } \\\n\text{    \text{ds})})])] \\\n\text{val})
\]

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

\[(\text{define-type} \text{ DefrdSub})\]
\[
\quad [\text{mtSub}]
\]
\[
\quad [\text{aSub} \ (\text{name symbol}?)]
\]
\[
\quad \ (\text{get-value} \ (\rightarrow \ \text{FAE-Value}?))
\]
\[
\quad \ (\text{rest} \ \text{DefrdSub}?))]
\]

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