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`

The following material won’t show up on any homework or exam
LC Grammar

\[
\langle LC \rangle ::= \langle id \rangle \\
| \{ \langle LC \rangle \hspace{1em} \langle LC \rangle \} \\
| \{ \text{fun} \hspace{1em} \{ \langle id \rangle \} \hspace{1em} \langle LC \rangle \} 
\]
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

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

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 \texttt{zero} as shorthand for the above expression:

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

Can you write a program that produces one?

```plaintext
    one  = {fun {f} {fun {x} {f x}}} = {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?

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

... 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?

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

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

true = def {fun {x} {fun {y} x}}
false = def {fun {x} {fun {y} y}}
zero = def {fun {f} {fun {x} x}}
one = def {fun {f} {fun {x} {f x}}}
two = def {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

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

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?

\[
\text{first} \quad \text{def} \quad \{\text{fun } \{p\} \quad \{p \ \text{true}\}\}
\]
\[
\text{rest} \quad \text{def} \quad \{\text{fun } \{p\} \quad \{p \ \text{false}\}\}
\]

Example:

\[
\{\text{first} \quad \{\text{cons zero} \ \text{one}\}\} \quad \Rightarrow \quad \{\text{first} \quad \{\text{fun } \{b\} \quad \{\{b \ \text{zero}\} \ \text{one}\}\}\}\}
\]
\[
\Rightarrow \quad \{\{\text{fun } \{b\} \quad \{\{b \ \text{zero}\} \ \text{one}\}\} \ \text{true}\}
\]
\[
\Rightarrow \quad \{\{\text{true} \quad \text{zero}\} \ \text{one}\}
\]
\[
\Rightarrow \quad \text{zero}
\]
Implementing Arithmetic

```plaintext
zero def = {fun {f} {fun {x} x}}
one def = {fun {f} {fun {x} {f x}}}
two def = {fun {f} {fun {x} {f {f x}}}}
```

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

```plaintext
add1 def = {fun {n}
    {fun {g} {fun {y}
        {g {{n g} y}}}}}
```

Example:

```plaintext
{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?

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

Implementing Arithmetic

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

```
add3  def = {fun {n} {add1 {add1 {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

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

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

two \overset{\text{def}}{=} \{ \text{fun } f \} \{ \text{fun } x \} f f x\}\}

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

\text{mult} \overset{\text{def}}{=} \{ \text{fun } n \} \{ \text{fun } m \} \{ \{ \text{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?

```plaintext
iszero 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?

```plaintext
shift  =  {fun {p} 
          {{cons {rest p}} {add1 {rest p}}}}

subl  =  {fun {n} 
          {first 
           {{n shift} {{cons zero} zero}}}}
```

And then subtraction is obvious...
Implementing Factorial

\[
\begin{align*}
\text{mk-rec} & \overset{\text{def}}{=} \{ \text{fun} \ \{ \text{body} \} \\
& \{ \{ \text{fun} \ \{ \text{fX} \} \ \{ \text{fX fX} \} \} \\
& \{ \text{fun} \ \{ \text{fX} \} \\
& \{ \{ \text{fun} \ \{ \text{f} \} \ \{ \text{body f} \} \} \\
& \{ \text{fun} \ \{ \text{x} \} \ \{ \{ \text{fX fX} \ \text{x} \} \} \} \} \}
\end{align*}
\]

Can you write a program that computes factorial?

\[
\{ \text{mk-rec} \\
\{ \text{fun} \ \{ \text{fac} \} \\
\{ \text{fun} \ \{ \text{n} \} \\
\{ \{ \text{iszero n} \} \\
\text{one} \} \\
\{ \{ \text{mult n} \ \{ \text{fac} \ \{ \text{sub1 n} \} \} \} \} \} \}
\]

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