Lambda Calculus

Your Favorite Language

Probably has lots of features:

- Assignment \( x = x + 1 \)
- Booleans, integers, characters, strings, ...
- Conditionals
- Loops
- return, break, continue
- Functions
- Recursion
- References / pointers
- Objects and classes
- Inheritance
- ...

Which ones can we do without?

What is the smallest universal language?
What is *computable*?  

*Alonzo Church*

**Before 1930s**

Informal notion of an effectively calculable function:

\[
\begin{array}{c|c}
\text{32} & \text{5512} \\
\text{32} & \text{32} \\
\text{231} & \text{224} \\
\text{72} & \text{} \\
\text{64} & \text{} \\
\text{8} & \text{}
\end{array}
\]

long division

\(5512 \div 32\)

can be computed by a human with pen and paper, following an algorithm
1936: Formalization

What is the smallest universal language?

Alan Turing

Church

Rube Goldberg contraption
Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.

Peter Landin, 1966

The Lambda Calculus

Has one feature:

- Functions
No, really

- Assignment ($x = x + 1$)
- Booleans, integers, characters, strings,…
- Conditionals
- Loops
- return, break, continue
- Functions
- Recursion
- References / pointers
- Objects and classes
- Inheritance
- Reflection

More precisely, **only thing** you can do is:
Define a function
Call a function

Describing a Programming Language

- **Syntax**: what do programs look like?
- **Semantics**: what do programs mean? “execute”
  - *Operational semantics*: how do programs execute step-by-step?
Syntax: What Programs Look Like

\[
e ::= x \\
\quad | \ x \rightarrow e \\
\quad | \ e_1 e_2
\]

Programs are expressions e (also called \( \lambda \)-terms) of one of three kinds:

- **Variable**
  - x, y, z

- **Abstraction** (aka nameless function definition)
  - \( \lambda x \rightarrow e \)
  - \( x \) is the **formal parameter**, \( e \) is the **body**
  - “for any \( x \) compute \( e \)”

- **Application** (aka function call)
  - \( e_1 e_2 \)
  - \( e_1 \) is the **function**, \( e_2 \) is the **argument**
  - in your favorite language: \( e_1(e_2) \)

(Here each of \( e \), \( e_1 \), \( e_2 \) can itself be a variable, abstraction, or application)
Examples

1. \( \lambda x \rightarrow x \)  
   -- The identity function (id)
   -- ("for any x compute x")

2. \( \lambda x \rightarrow (\lambda y \rightarrow y) \)  
   -- A function that returns (id)

3. \( \lambda f \rightarrow (f \, \lambda x \rightarrow x) \)  
   -- A function that applies its argument to \( \lambda x \rightarrow x \)

---

QUIZ

Which of the following terms are syntactically incorrect?

\( \text{const id} = (\lambda x \rightarrow x) \)  
\( \text{const bob} = (\lambda x \rightarrow \text{id}) \)

---

NOT VALID LC EXPRESSIONS
A. \( (\lambda x \rightarrow x) \rightarrow y \) not valid

B. \( \lambda x \rightarrow x \cdot x \)

C. \( (\lambda x \rightarrow x \cdot (y \cdot x)) \)

D. A and C

E. all of the above

\( ((x \cdot y) \neq) \)

\( (x \cdot (y \neq)) \)

Examples

\( \lambda x \rightarrow x \) -- The identity function
   -- ("for any x compute x")

\( \lambda x \rightarrow (\lambda y \rightarrow y) \) -- A function that returns the identity function

\( \lambda f \rightarrow f ((\lambda x \rightarrow x)) \) -- A function that applies its argument to the identity function
How do I define a function with two arguments?

- e.g. a function that takes \( x \) and \( y \) and returns \( y \)?

\[
(x, y) \Rightarrow y
\]

\[
(\lambda x \rightarrow (\lambda y \rightarrow y))
\]

\[
f \; x_1 \; x_2 \; x_3
\]

\[
((f \; x_1) \; x_2 \; x_3) \quad f \; (x_1 \; (x_2 \; x_3))
\]

\[
\lambda x \rightarrow (\lambda y \rightarrow y)
\]

-- A function that returns the identity function

-- OR: a function that takes two arguments

-- and returns the second one!
How do I apply a function to two arguments?

- e.g. apply \( \lambda x \to (\lambda y \to y) \) to apple and banana?

\[
\begin{aligned}
(\lambda x \to (\lambda y \to y))
&= ((\text{func \ apple}) \text{ banana})
\Rightarrow \text{ banana}

\frac{\lambda y t \to e}{\equiv (\lambda x \to (\lambda y \to (\lambda z \to e)))}
\end{aligned}
\]

\(((\lambda x \to (\lambda y \to y)) \text{ apple}) \text{ banana}) \text{ -- first apply to apple,}

\text{ -- then apply the result to banana}

o \text{ banana}