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?

Before 1930s

Informal notion of an **effectively calculable** function:

![Division Calculation](image)

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

What is the smallest universal language?

Alan Turing

21 yrs
The Next 700 Languages
Peter Landin

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 \( x \rightarrow x + 1 \)
- 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?
  - **Operational semantics**: how do programs execute step-by-step?

Syntax: What Programs Look Like

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

- **Variable**
  - \( x, y, z \)
• **Abstraction** (aka _nameless_ function definition)

  - \(\lambda x \to e\)
  - \(x\) is the _formal_ parameter, \(e\) is the _body_
  - “for any \(x\) compute \(e\)”

  \[ \text{function } (x) \{ \text{return } e^3 \} \]

• **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**

\[ \text{function } (x) \{ \text{return } x^3 \} \]
\( \lambda x \to x \) -- The identity function

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

\( \lambda f \to f (\lambda x \to x) \) -- A function that applies its argument

\( \lambda x \to (\lambda y \to y) \) \text{ return } y \}

\( \lambda x \to e \]

**QUIZ**

Which of the following terms are syntactically **incorrect**?

A. \((\lambda x \to x) \to y\) \text{ NOT valid LC exprs}

B. \( \lambda x \to x \to x \) \checkmark

C. \( \lambda x \to x (y \to x) \) \checkmark \( e_1(e_2) \)

D. A and C
\( x \rightarrow (x \cdot y) \cdot z \)

\( x \rightarrow (x \cdot y) \cdot z \)

\( (((x \cdot y) \cdot z) \cdot a) \cdot b \)

**Examples**

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

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

\( f \rightarrow f (x \rightarrow x) \)  
-- 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 \( y \)?
How do I apply a function to two arguments?
• e.g. apply \( x \rightarrow (y \rightarrow y) \) to apple and banana?

\[
\text{function}(x) \equiv \text{return function}(y) \equiv \text{return } y; \text{if}
\]

\[
((x \rightarrow (y \rightarrow y)) \text{ apple}) \text{ banana} \quad -- \text{first apply to apple,}
\]

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

**Syntactic Sugar**
instead of | we write
---|---
\(\lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow e))\) | \(\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e\)
\(\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e\) | \(\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e\)
\(((\lambda e1 \lambda e2 \lambda e3 \lambda e4)\) | \((\lambda e1 \lambda e2 \lambda e3 \lambda e4)\)  

\(\lambda x \rightarrow \lambda y \rightarrow y\)  
--- A function that takes two arguments  
--- and returns the second one...  

\((\lambda x \rightarrow y)\) apple banana  
--- applied to two arguments
How do I “run” / “execute” a λ-term?

Think of middle-school algebra:

-- Simplify expression:

\[(x + 2) \times (3x - 1)\]

= 

???

Execute = rewrite step-by-step following simple rules, until no more rules apply

Rewrite Rules of Lambda Calculus
1. $\alpha$-step (aka renaming formals)
2. $\beta$-step (aka function call)

But first we have to talk about **scope**

**Semantics: Scope of a Variable**

The part of a program where a variable is visible.

In the expression $\lambda x \rightarrow e$:

- $x$ is the newly introduced variable
- $e$ is the **scope** of $x$
- any occurrence of $x$ in $\lambda x \rightarrow e$ is **bound** (by the binder $\lambda x$)

For example, $x$ is bound in:

1. $\lambda x \rightarrow x$
2. $\lambda x \rightarrow (\lambda y \rightarrow x)$
An occurrence of $x$ in $e$ is free if it's *not bound* by an enclosing abstraction.

For example, $x$ is free in:

```
\( \lambda y \cdot \lambda y \cdot x \) -- no binders at all!
\( \lambda x \cdot \lambda y \cdot \lambda x \) -- no $|x$ binder
\( \lambda x \cdot \lambda y \cdot \lambda y \cdot x \) -- $x$ is outside the scope of the $|x$ binder;
```

**Quiz**

In the expression $\lambda x \cdot \lambda x \cdot x$, is $x$ bound or free?

A. bound

B. free
Free Variables

An variable \( x \) is free in \( e \) if there exists a free occurrence of \( x \) in \( e \)

We can formally define the set of all free variables in a term like so:

\[
\begin{align*}
FV(x) & = ??? \\
FV(\setminus x \rightarrow e) & = ??? \\
FV(e1 \ e2) & = ???
\end{align*}
\]
Closed Expressions

If e has no free variables it is said to be closed

- Closed expressions are also called combinators

What is the shortest closed expression?

\( \hat{x} \rightarrow x \)
Rewrite Rules of Lambda Calculus

1. $\alpha$-step (aka renaming formals)
2. $\beta$-step (aka function call)

"Copy-paste"

Semantics: $\beta$-Reduction
where $e_1[x := e_2]$ means “$e_1$ with all free occurrences of $x$ replaced with $e_2$”

Computation by search-and-replace:

- If you see an abstraction applied to an argument, take the body of the abstraction and replace all free occurrences of the formal by that argument.

- We say that $(\lambda x \rightarrow e_1) e_2$ $\beta$-steps to $e_1[x := e_2]$

  $$(\lambda x \rightarrow x) \text{apple} = \beta \rightarrow \text{apple}$$

  $$(\lambda x \rightarrow (\lambda y \rightarrow y)) \text{apple} \text{banana}$$

  $$(\lambda x \rightarrow (\lambda y \rightarrow y)) \text{banana}$$

Examples

$$(\lambda x \rightarrow x) \text{apple}$$

$= \beta \rightarrow \text{apple}$$
Is this right? Ask Elsa (http://goto.ucsd.edu:8095/index.html#?demo=blank.lc)!

\[
(f \rightarrow f (x \rightarrow x)) \text{ (give apple)}
\]

\[
(b > ???)
\]

\[
(\lambda x \rightarrow (\lambda y \rightarrow y)) \text{ apple} \rightarrow^* (\lambda y \rightarrow y)
\]

\[
(\lambda y \rightarrow (\lambda y \rightarrow y)) \text{ apple} \rightarrow^* ???
\]

A. \( \lambda y \rightarrow y \)

B. \( \lambda y \rightarrow \text{apple} \)

C. \( \lambda x \rightarrow \text{apple} \)

D. \( \lambda x \rightarrow (x \rightarrow x)(\text{apple}) \)

\[
\text{QUIZ}
\]

\[
(\lambda x \rightarrow (\lambda y \rightarrow x)) \text{ apple} \rightarrow^* (\lambda y \rightarrow y)
\]

\[
>b> (x \rightarrow x)[x:= \text{apple}]
\]

A. apple

B. \( \lambda y \rightarrow \text{apple} \)

C. \( \lambda x \rightarrow \text{apple} \)