Lambda Calculus

Your Favorite Language

Probably has lots of features:

- Assignment (`x = x + 1`)
- Booleans, integers, characters, strings, ... arrays
- 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**?

CANVAS
- github USER

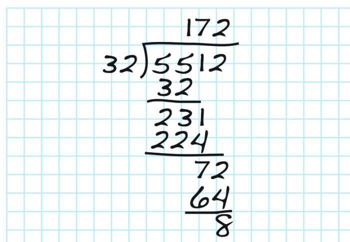
GITHUB
- do assignment
00-lambda

What is computable?

TURING MACHINE

Before 1930s

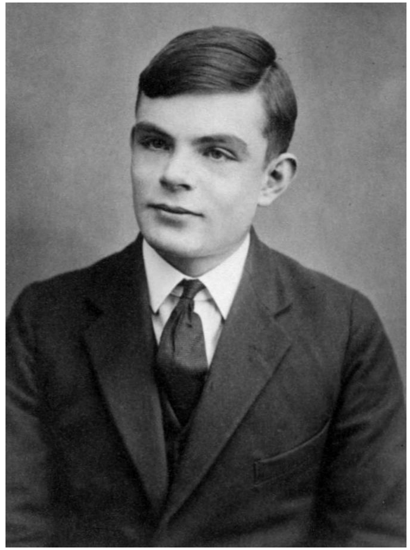
Informal notion of an **effectively calculable** function:



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

1936: Formalization

What is the **smallest universal language**?



21 yrs

Alan Turing

Alonzo Church

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 = 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

SEMANTICS

Syntax: What Programs Look Like

Expr

variables

func

call

```
e ::= x
    | \x -> e
    | (e1 e2)
```

function(x){return e}

$e_1(e_2)$

Programs are **expressions** `e` (also called **λ-terms**) of one of three kinds:

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

- **Abstraction** (aka *nameless* function definition)
  - `\x -> e` (x => e)
  - `x` is the *formal* parameter, `e` is the *body*
  - "for any `x` compute `e`" function (x) {return e}
- **Application** (aka function call)
  - $\boxed{\texttt{e1 e2}}$
  - `e1` is the *function*, `e2` is the *argument*
  - in your favorite language: `e1(e2)`

(Here each of `e`, `e1`, `e2` can itself be a variable, abstraction, or application)

Examples

function (x) {return x}

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

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

\f -> f (\x -> x)     -- A function that applies its argument
                      -- to the identity function
```

in

out

function (x) { return (function(y) {return y}); }

$e ::= x \mid \backslash x \to e \mid (e_1\ e_2)$

QUIZ

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

A. `\(\x -> x) -> y` ✗

B. `\x -> x x` ✓

C. `\x -> x (y x)` ✓

D. A and C

NOT valid LC exprs

$e_1(e_2)$

E. all of the above

\x → ((x y) x)

\x → ((x y) x)

((((x y) z) a) b)

Examples

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

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

\f -> f (\x -> 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` ?

takes input x
returns a second funct
takes input y
return y.

```
\x -> (\y -> y)  -- A function that returns the identity function
                 -- OR: a function that takes two arguments
                 -- and returns the second one!
```

port to JS

"call with 2 inputs"
to see result

How do I apply a function to two arguments?

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

\x → (\y → y)

function(x){ return function (y) { return y; };}

```
(((\x -> (\y -> y)) apple) banana) -- first apply to apple,
                                   -- then apply the result to banana
```

Syntactic Sugar

instead of	we write
`\x -> (\y -> (\z -> e))`	`\x -> \y -> \z -> e`
`\x -> \y -> \z -> e`	`\x y z -> e`
`(((e1 e2) e3) e4)`	`e1 e2 e3 e4`

\x y z -> e

```
\x y -> y      -- A function that that takes two arguments
               -- and returns the second one...

(\x y -> y) apple banana -- ... applied to two arguments
```

Semantics : What Programs Mean

How do I "run" / "execute" a λ-term?

Think of middle-school algebra:

```
-- Simplify expression:

  (x + 2)*(3*x - 1)
 =
  ???
```

$$(2+3) * (9-4)$$
$$\Downarrow$$
$$(2+3) * 5$$
$$\Downarrow$$
$$5 * 5$$
$$\Downarrow$$
$$25$$

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

Rewrite Rules of Lambda Calculus

1. α-step (aka *renaming formals*)
2. β-step (aka *function call*)

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

Semantics: Scope of a Variable

function (x, y) {

// BODY

"scope of x, y"

}

The part of a program where a **variable is visible**

In the expression `\x -> e`

function (x) { // scope x }

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

For example, `x` is bound in:

```
① \x -> x
② \x -> (\y -> x)
```

An occurrence of `x` in `e` is **free** if it's *not bound* by an enclosing abstraction

For example, `x` is free in:

```
x y                -- no binders at all!
\y -> x y          -- no \x binder
(\x -> \y -> y) x  -- x is outside the scope of the \x binder;
                   -- intuition: it's not "the same" x
```

fun(x){return fun(y){ret y;}} (x)

NOT AN "OCCURENC"

	1	2
A	B	B
B	B	F
C	F	B
D	F	F

QUIZ

In the expression `(\x -> x) x`, is `x` *bound* or *free*?

A. bound

B. free

~~**C.** first occurrence is bound, second is free~~

~~**D.** first occurrence is bound, second and third are free~~

~~**E.** first two occurrences are bound, third is free~~

(\x -> x) x B F

((\x -> x x)) B B

(\x -> x x)

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:

```
FV(x)         = ???
FV(\x -> e)   = ???
FV(e1 e2)     = ???
```

{apple}

$\boxed{a\ (\backslash x \to x)}$

FV ?

x, a

(A) {}

(B) {a}

(C) {x}

(D) {a, x}

$(\backslash a \to a)\ a$

① ②

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?

$\backslash x \to x$

$\cancel{\backslash x \to y}$

Rewrite Rules of Lambda Calculus

1. α-step (aka *renaming formals*)
2. β-step (aka *function call*)

"Copy-paste"

\x -> x) apple =b> apple

x e1 e2

Semantics: β-Reduction

(\x -> e1) e2

```
(\x -> e1) e2 =b> e1[x := e2]
```

where `e1[x := e2]` means " `e1` with all *free* occurrences of `x` replaced with `e2` "

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 `(\x -> e1) e2` β-steps to `e1[x := e2]`

(\x → x) apple =b> apple

forma e1 e2

$e_1[x := e_2]$

((\x → (\y → y)) apple) banana

"\x" e_1 e_2

=b> (\y → y) banana

"x" "e_1" "e_2"

=b> banana

Examples

```
(\x -> x) apple
=b> apple
```

Is this right? Ask Elsa (http://goto.ucsd.edu:8095/index.html#?demo=blank.lc)!

```
(\f -> f (\x -> x)) (give apple)
=b> ???
```

$(\backslash x \to (\backslash y \to y))\ apple \overset{b}{\rightsquigarrow} (\backslash y \to y)$

$(\backslash y \to (\backslash y \to y))\ apple \overset{b}{\rightsquigarrow}$???

$(\backslash x \to e_1)\ e_2 \overset{b}{\rightsquigarrow} e_1[x := e_2]$

(A) \y -> y ✓

(B) \y -> apple

(C) ~~\apple -> y~~ ✗

(D) ~~\apple -> apple~~ ✗

$(\backslash x \to (x\ x))\ apple$

QUIZ

$=b> (x\ x)[x := apple]$

(apple apple)

```
(\x -> (\y -> y)) apple
=b> ???
```

x, e_1, e_2

$e_1[x := e_2]$

$(\backslash y \to y)$

A. apple

B. \y -> apple

C. \x -> apple