Lambda Calculus

Your Favorite Language

Probably has lots of features:

- Assignment (`x = x + 1`) ✓
- Booleans, integers, characters, strings, ... datatypes
- Conditionals - if-then-else
- Loops - for / while
- `return`, `break`, `continue`
- Functions
- Recursion
- References / pointers 1930s
- Objects and classes
- Inheritance
- ...

Which ones can we do without?

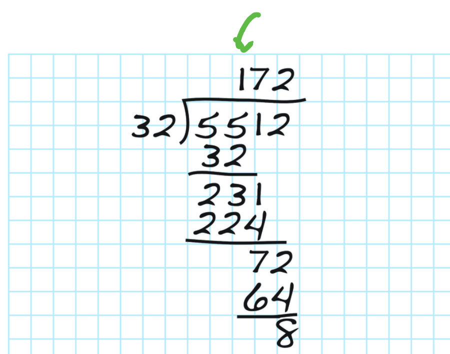
What is the **smallest universal language**?

What is computable?

Alonzo Church

Before 1930s

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



long division

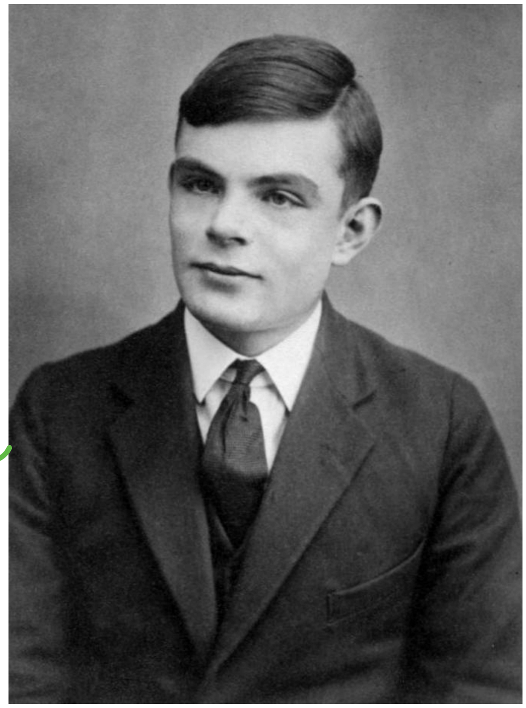
5512/82

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

1936: Formalization

What is the **smallest universal language**?

Rube Goldberg Contraption



TURING MACHINE

Alan Turing

Church

> *Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.*

Peter Landin, 1966

→ AWS - LAMBDA

2015

The Lambda Calculus

Has one feature:

- Functions /Call

No, *really*

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

① Bcoz its cool

② Haskell / FP

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

({ }) =>

LC JS

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

$e_1(e_2)$

x, y, z

function(x){return e;}

(x) → {return e;}

(x) → e

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

- **Variable**
  - x, y, z apple zoom voltron

(e)

- **Abstraction** (aka *nameless* function definition)
  - \x -> e
  - x is the *formal* parameter, e is the *body*
  - "for any x compute e"
- **Application** (aka function call)
  - e1 e2 $e_1(e_2)$
  - 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

(x) => x ≈ (y) => y

1

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

≈

2

```
\x -> (\y -> y)          -- A function that returns (id)
```

3

```
\f -> (f (\x -> x))      -- A function that applies its argument t
o id
```

const id = (x) => x;

const bob = (x) => id;

$\underbrace{\backslash f \to f}_{e_1}\ \underbrace{\backslash x \to x}_{e_2}$

$\boxed{\backslash f \to \boxed{f\ \boxed{\backslash x \to x}}}$

QUIZ

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

NOT VALID λC EXPRESSIONS

A. `\(\x -> x) -> y` not valid

B. `\x -> x x`

C. `(\x -> x (y x))`

D. A and C

E. all of the above

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

((x y) z)

apple

(x (y z))

Examples

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

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

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

How do I define a function with two arguments?

$(x, y) \Rightarrow y$

- e.g. a function that takes `x` and `y` and returns `y`?

$(\backslash x \to (\backslash y \to y))$

$f \;\; x_1 \;\; x_2 \;\; x_3$

LEFT RIGT

$(((f\ x_1)\ x_2)\ x_3)$ $\quad$ $f\ (x_1\ (x_2\ x_3))$

```
\x -> (\y -> 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 `(\x -> (\y -> y))` to `apple` and `banana`?

((func ap) ban)

$\underbrace{(\backslash x \to (\backslash y \to y))}$ apple banana

((func apple) banana) $\Rightarrow$ banan

$\boxed{\backslash x y z \to e}$ $\Rightarrow$ $\underbrace{(\backslash x \to (\backslash y \to (\backslash z \to e)))}$

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