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?

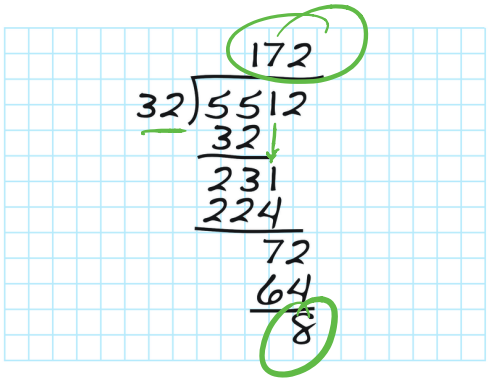
What is the **smallest universal language**?

What is computable?

Turing Machine (105)
LC

Before 1930s

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



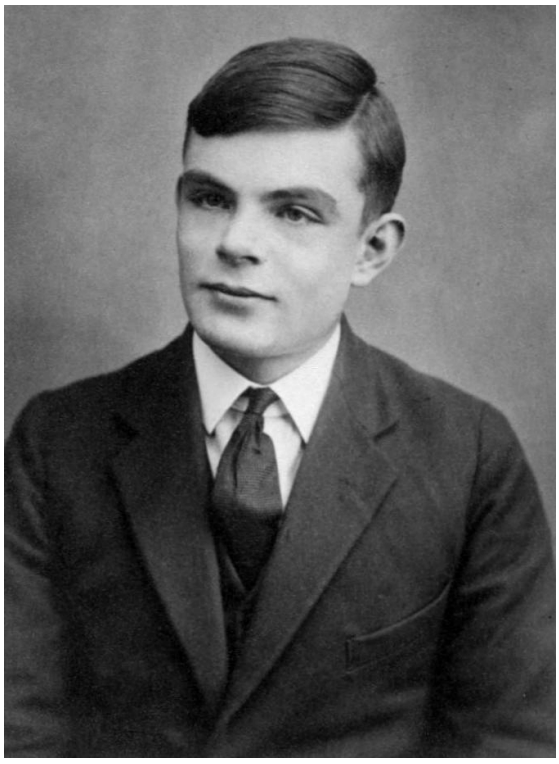
can be computed by a human with pen and paper, following an algorithm

1936: Formalization

What is the **smallest universal language**?

Turing

105



Alan Turing

1930s

Alonzo Church

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 / Function Calls

No, *really*

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

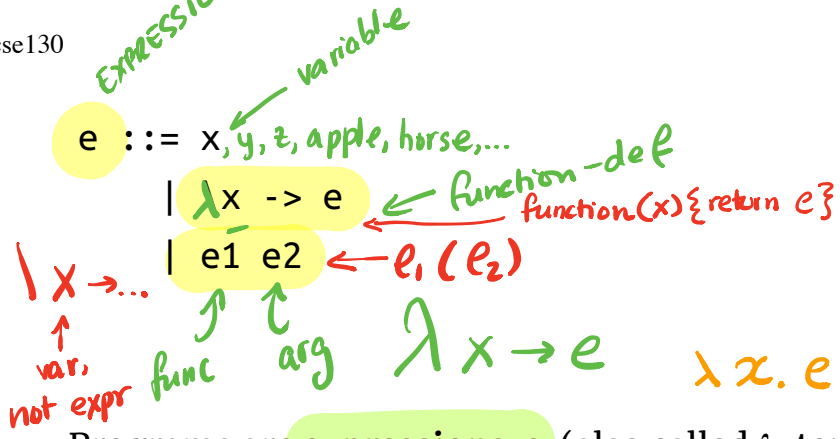
More precisely, *only thing* you can do is:

- **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?

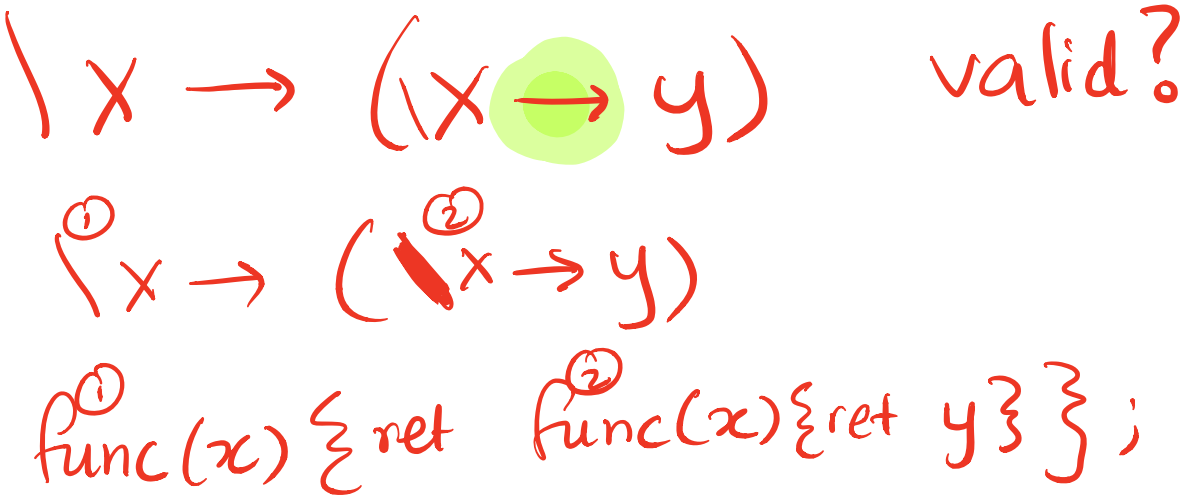
Syntax: What Programs Look Like



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` is the *formal* parameter, `e` is the *body*
  - "for any `x` compute `e`"
- **Application** (aka function call)
  - `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

① function(y) { return y } ... "id"

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

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

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

② function (x) { return function (y) { return y }; };

((x y) z)

(x (y z))

① x (y (z))

② (x (y)) (z)

QUIZ

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

A. `\(\x -> x) -> y` ? ✓(A) function (x) { return x (x) }

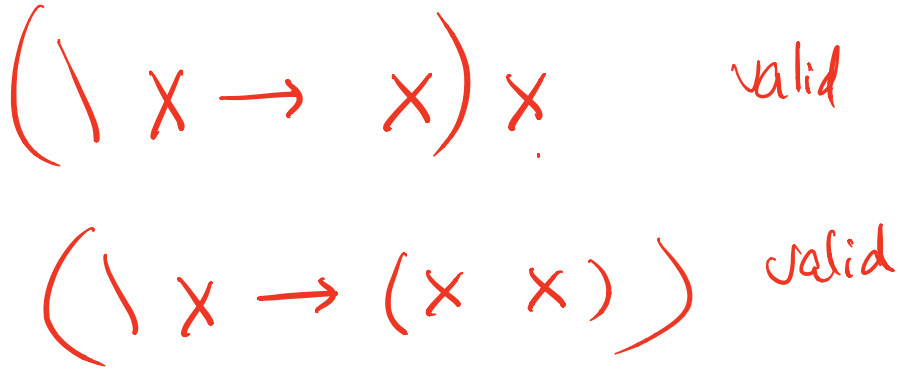
B. `\x -> (x x)` ✗(B) function (x) { return func (x) { return x } }

C. `\x -> x (y x)` → func (x) { ret x (y (x)) };

D. A and C

E. All of the above

E. all of the above

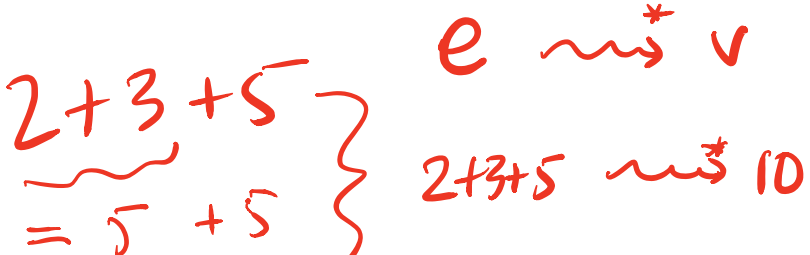


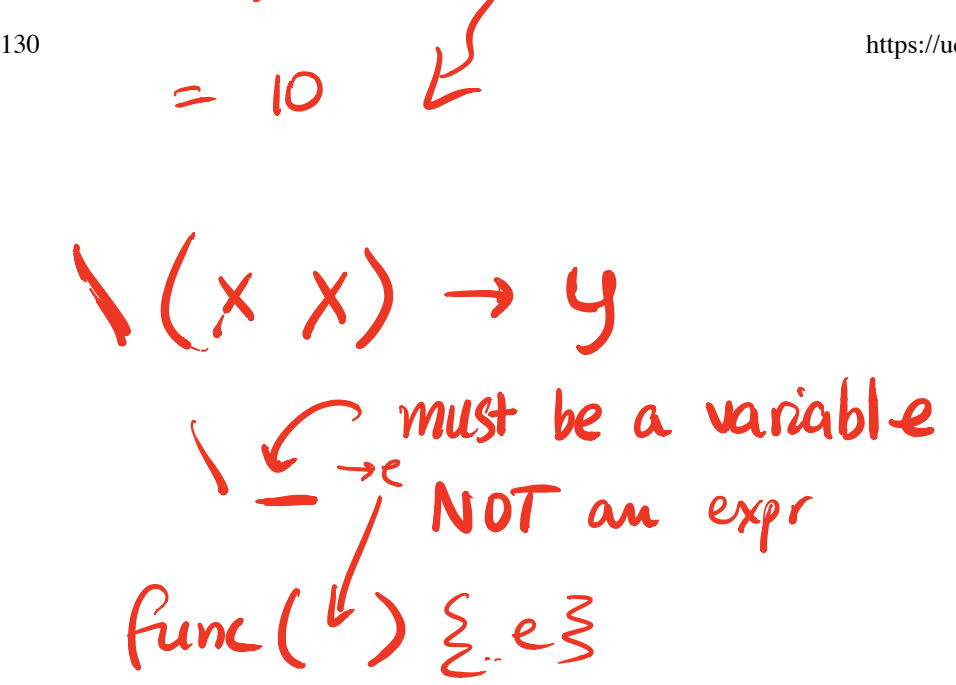
Examples

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

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

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

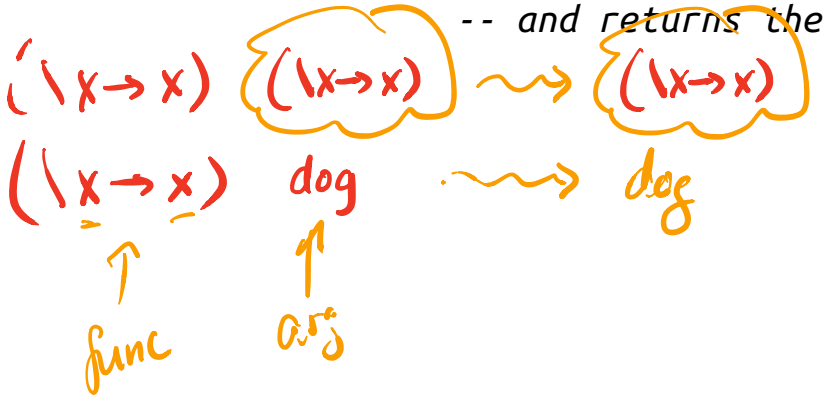
How do I define a function with two arguments?

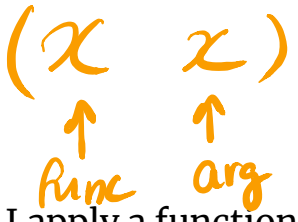
- e.g. a function that takes `x` and `y` and returns `y`?






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

                    -- 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`?