Lambda Calculus

Your Favorite Language

Probably has lots of features:

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

Which ones can we do without?

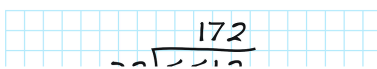
What is the **smallest universal language**?

1921

What is computable?

Before 1930s

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



Alan Turing

Lambda Calculus

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

function(x) { BODY }

$e_1 (e_2)$

x

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

expr

```
e ::= x, y, z, ...
   | \x -> e
   | e1 e2
```

variable

lambda x: e

(x) => e

function(x){ return e }

|x| { e }

$e_1(e_2)$

call

func

arg

anon. func def

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

function(x) { retorn x }

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

input result

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

expr

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

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

Func(f) { return f(func (x){return x}) }

QUIZ

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

NOT valid LC Programs

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

B. `\x -> x x`

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

D. A and C

E. all of the above

\x → x (y x)

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`?

(\x -> (\y -> y))

```
\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` ?

e_1 e_2 e_3

((\x -> (\y -> y)) apple) banana)

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

((x y) z)

$((((e_1\ e_2)\ e_3)\ e_4)\ e_5)$

$(\backslash x_1 \to (\backslash x_2 \to (\backslash x_3 \to (\backslash x_4 \to e))))$

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

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