

# 1 The Lambda Calculus

Most programming languages have modern features like

- Assignments
- Types
- Conditions
- Loops
- Classes
- And so on. . .

However the smallest universal language doesn't need *any of these* at all.

## 1.1 The Smallest Universal Language

What is the smallest universal language?

### 1.1.1 What is Computable?

Before the 1930s, there was the informal notion of an *effectively calculable* function. That is, it can be computed by a human with pen and paper, followed by an algorithm.

### 1.1.2 Formalization of a Language

**Alan Turing** introduced the *Turing Machine*, which is an infinite tape with some symbols, a head that can read from and write to the tape. To actually interact with a Turing machine, it has a state machine which has a bunch of states, along with a transition function, which tells it how to move around and what to do. The programming language is essentially the transition function of the Turing machine.

**Alonzo Church** came up with the *Lambda Calculus*, which is (in some sense) simpler than a Turing machine.

**Peter Landin** used the Lambda Calculus to formalize the notion of a programming language. Lambda Calculus was influential in the creation of many modern programming languages, especially functional programming languages like Haskell.

### 1.1.3 The Lambda Calculus

It has one feature: *functions*. It does not have assignments, primitive types, control flow, recursion, etc. It literally only has *functions*. Specifically, you can

- define a function.
- call a function.

### 1.1.4 Describing a Programming Language

We're interested in two things:

- Syntax: what do programs look like? We use formal grammars (context-free grammars) to explain the syntax of a programming language.
- Semantics: what do programs mean? Specifically, *operational semantics*, or the idea of how programs execute step-by-step.

## 1.2 Syntax

We have one syntactical category: expressions (also called  $\lambda$ -terms).

$$\underbrace{E}_{\text{Expression}}$$

What can this expression expand to?

$$E ::= \underbrace{x}_{\text{Variable}} \mid \underbrace{\lambda x \rightarrow E}_{\text{Abstraction}} \mid \underbrace{E_1 E_2}_{\text{Application}}$$

The expression  $E$  can be one of three kinds:

- Variables:  $x$  can be any variable, e.g.  $y$  or  $z$  or even something like **apple**. We should think of variables as mathematical variables (immutable). It does not change its value over time; it's like a variable in math where all it does is holds its value.
- Abstraction: We can think of  $\lambda x \rightarrow E$  in the following mathematical way:

$$f(x) = E$$

This is, specifically, a **nameless function** that takes in input  $x$  (i.e. formal parameter) and returns an expression  $E$  (i.e. the body).

- Application: Here,  $E_1$  is the function and  $E_2$  is the argument. We can think of this as  $E_1(E_2)$  in any programming language. In mathematics, we might write  $f(5)$ . In Lambda Calculus, this would be **f 5**.

Note that  $E$ ,  $E_1$ , and  $E_2$  can itself be a variable, abstraction, or application. Additionally, note that application binds stronger than abstractions<sup>1</sup>.

Effectively, in Lambda Calculus, functions are *first-class*: we can apply functions to other functions, and a function can return another function.

## 1.3 Examples

We will now go over some examples.

- Consider the following program.

**apple**

This does nothing, but is a syntactically value program in Lambda Calculus.

- Consider the following program.

**apple banana**

This is an *application* of the variable **apple** to the variable **banana**.

- Consider the following program.

**$\lambda x \rightarrow x$**

---

<sup>1</sup>This will be important later on when we have syntactical sugar.

The identity function, which says that *for any  $x$ , compute  $x$* . This is the first program which is meaningful for us; it is a *very* important function.

- Consider the following program.

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

This (whole program) is an application of the identity function to the variable `apple`. The result of this program will be `apple`. Note that `()` is not in the grammar that we described above; the grammar is known as an *abstract syntax*, which simply describes what the programming language should have. In other words, the parentheses are ignored. Note that if we have

```
\x -> x apple
```

which is a different program. Here, `x apple` is the body.

- Consider the following program.

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

Here, we introduce another variable `y`. This takes one argument `x`, completely ignores `x`, and returns the identity function. Comparing this to the previous example, all we're doing is changing the name of the formal parameter.

- Consider the following program.

```
\f -> f (\x -> x)
```

All this does is takes a argument `f`, and applies that argument to the identity function.

## 1.4 Two Input Arguments

Suppose you wanted a function that takes arguments `x` and `y` and returns `y`. How would you implement this?

Consider the following program.

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

Here, this function returns the identity function. This is the same thing as a function that takes two arguments and returns the second one.

### 1.4.1 Applying Function to Two Arguments

For example, how do we apply `\x -> (\y -> y)` to `apple` and `banana`?

We can do something like

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

This first applies `apple` and then applies to `banana`.

## 1.5 Syntactical Sugar

The following are syntactical sugar for common operations.

Instead of	We can write
$\backslash x \rightarrow (\backslash y \rightarrow (\backslash z \rightarrow E))$	$\backslash x \rightarrow \backslash y \rightarrow \backslash z \rightarrow E$
$\backslash x \rightarrow \backslash y \rightarrow \backslash z \rightarrow E$	$\backslash x \ y \ z \rightarrow E$
$((E1 \ E2) \ E3) \ E4$	$E1 \ E2 \ E3 \ E4$

Additionally, the following are equivalent:

```

\ a -> (\ b -> b)
\ a -> \ b -> b
\ a b -> b

```

The parentheses are applied starting from the right side.