1 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. **What is the smallest universal language?**

1.1 The Smallest Universal Language

1.1.1 What is Computable?

Before the 1930s, therw 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.

1.2 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.

1.2.1 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.

1.2.2 Syntax

We have one syntaxtical category: expressions (also called λ-terms).

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

What can this expression expand to?

$$\texttt{E ::= } \underbrace{\texttt{x}}_{\text{Variable}} \texttt{ | } \overbrace{\texttt{\textbackslash x -> E}}^{\text{Abstraction}} \texttt{ | } \underbrace{\texttt{E1 E2}}_{\text{Application}}$$

Note that `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.

We can think of `x -> E` in the following mathematical way:

$$f(x) = E$$

This is, specifically, a nameless function that takes in input x and returns an expression E.

For calling functions, in math, we do something like $f(5)$. In Lambda Calculus, we do `f 5`.

1.2.3 Examples

Consider the following program.

```
apple
```

This does nothing, but is a syntaxtically 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.

```
\x -> x
```

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 parenthesis 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.

1.2.4 Two Input Arguments

Suppose you wanted a function that takes arguments `x` and `y` and returns `y`?

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.

1.2.5 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`.

1.2.6 Syntatical Sugar

- Instead of

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

  we can write

  ```
  \x -> \y -> \z -> E
  ```

- Instead of

  ```
  \x -> \y -> \z -> E
  ```

  we can write

  ```
  \x y z -> E
  ```

- Instead of

  ```
  (((E1 E2) E3) E4)
  ```

  we can write

  ```
  E1 E2 E3 E4
  ```