

# 1 Lambda Calculus

## 1.1 Programming in Lambda Calculus

How do we encode features like

- Booleans
- Records
- Numbers
- Recursion

and more?

### 1.1.1 Booleans

How can we encode Boolean values, like `TRUE` or `FALSE`, as functions?

To answer this question, we ask another one: what *do* we do with a Boolean `b`? Making a binary choice is one:

```
if b then E1 else E2
```

We want to define three functions

```
let TRUE = ???
let FALSE = ???
let ITE = \b x y -> ???      -- if b then x else y
```

such that

```
ITE TRUE apple banana =~> apple
ITE FALSE apple banana =~> banana
```

Our implementation is as follows:

```
let TRUE = \x y -> x          -- Returns the first argument
let FALSE = \x y -> y         -- Returns the second argument
let ITE = \b x y -> b x y     -- Applies condition to branch
```

To see how this works, suppose we want to evaluate `ITE TRUE egg ham`, which should resolve to `egg`. We have:

```
eval ite_true:
  ITE TRUE egg ham
=d> (\b x y -> b x y) TRUE egg ham    -- Expand ITE
=b> (\x y -> TRUE x y) egg ham         -- Beta-step on TRUE
=b> (\y -> TRUE egg y) ham             -- Beta-step on egg
=b> TRUE egg ham                      -- Beta-step on ham
=d> (\x y -> x) egg ham                -- Expand TRUE
=b> (\y -> egg)                        -- Beta-step on egg
=b> egg                               -- Beta-step on ham
```

### 1.1.2 Boolean Operators

Now that we have TRUE, FALSE, and ITE, we can define other Boolean operators like:

```
let NOT = \b      -> ???
let AND = \b1 b2 -> ???
let OR  = \b1 b2 -> ???
```

Recall that:

$$\text{NOT}(b) = \begin{cases} \text{FALSE} & b \text{ is TRUE} \\ \text{TRUE} & b \text{ is FALSE} \end{cases}$$

$$\text{AND}(b_1, b_2) = \begin{cases} \text{TRUE} & b_1 \text{ is TRUE and } b_2 \text{ is TRUE} \\ \text{FALSE} & \text{Otherwise} \end{cases}$$

$$\text{OR}(b_1, b_2) = \begin{cases} \text{TRUE} & b_1 \text{ is TRUE or } b_2 \text{ is TRUE} \\ \text{FALSE} & \text{Otherwise} \end{cases}$$

The implementation is as follows:

```
let NOT = \b      -> ITE b FALSE TRUE
                    -> b FALSE TRUE

let AND = \b1 b2   -> ITE b1 (ITE b2 TRUE FALSE) FALSE
                    -> ITE b1 b2 FALSE
                    -> b1 b2 FALSE

let OR  = \b1 b2   -> ITE b1 TRUE b2
                    -> b1 TRUE b2
```

### 1.1.3 Records

A record (tuple) is a way to bundle multiple values together and then access them. The simplest kind of a record is a **pair**, which holds two values. In particular, a pair can:

- Pack two items into a pair.
- Get the first item.
- Get the second item.

We need to implement the following functions:

```
let MKPAIR = \x y -> ???      -- Makes a pair with elements x and y
let FST    = \p  -> ???      -- Returns the first element of the pair p
let SND    = \p  -> ???      -- Returns the second element of the pair p
```

The functions work like so:

```
FST (MKPAIR apple banana) =~> apple
SND (MKPAIR apple banana) =~> banana
```

One thing to notice is that we can use a *boolean* value to indicate whether we want the first or second element. So, creating a pair is the same thing as creating a function which returns value *x* or *y* based on whether TRUE or FALSE is passed in.

```
let MKPAIR = \x y -> (\b -> ITE b x y)
let FST    = \p  -> p TRUE      -- Returns the first element
let SND    = \p  -> p FALSE     -- Returns the second element
```

Now, suppose we want to make a triple  $(x, y, z)$ . The idea is that we can make use of two pairs, like so:  $((x, y), z)$ . We can define our implementation like so:

```
let MKTRIPLE = \x y z -> MKPAIR (MKPAIR x y) z
let FST3     = \t      -> FST (FST t)
let SND3     = \t      -> SND (FST t)           -- Apply FST to t first
                                                -- and then apply SND to FST t
let TRD3     = \t      -> SND t
```

Alternatively, if we have  $(x, (y, z))$ , our implementation will be:

```
let MKTRIPLE = \x y z -> MKPAIR x (MKPAIR y z)
let FST3     = \t      -> FST t
let SND3     = \t      -> FST (SND t)
let TRD3     = \t      -> SND (SND t)
```

### 1.1.4 Numbers

Let us start with natural numbers  $\{0, 1, 2, \dots\}$ . What can we do with natural numbers?

- We can count them: 0, inc.
- Arithmetic: dec, +, -, \*.
- Comparisons: ==, <=, etc.

We need to define the following:

- A family of **numerals**: ZERO, ONE, TWO, THREE.
- Arithmetic functions: INC, DEC, ADD, SUB, MULT.
- Comparisons: IS\_ZERO, EQ.

Where they respect all regular laws of arithmetic; for example:

```
IS_ZERO ZERO      =~> TRUE
IS_ZERO (INC ZERO) =~> FALSE
INC ONE           =~> TWO
.
.
.
```

How do we implement numerals? We can define a numeral as the number of times we can apply a function. In particular, we define a **church numeral** as a number  $N$  which is encoded as an combinator that calls a function on an argument  $N$  times.

```
let ZERO  = \f x -> x
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let THREE = \f x -> f (f (f x))
let FOUR  = \f x -> f (f (f (f x)))
let FIVE  = \f x -> f (f (f (f (f x))))
let SIX   = \f x -> f (f (f (f (f (f x))))))
.
.
.
```