# Lambda Calculus in Elixir

Bernardo Amorim

Let’s play a game:

Guess The Programming Language

# Similarities?

Lambda

# Functional Programming Languages


λ ≈ FP ?

# Anonymous Functions

## Python

```
lambda x: x
```

## Ruby

```
lambda { |x| x }
```

# Alonzo Church

## AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By Alonzo Church.

**1. Introduction.** There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function $f$ of $n$ positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer $n$ whether or not there exist positive integers $x, y, z$, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function $f$, such that $f(n)$ is equal to 2 if and only if there exist positive integers $x, y, z$, such that $x^n + y^n = z^n$. Clearly the condition that the function $f$ be effectively calculable is an essential part of the problem, since without it the problem becomes trivial.

Another example of a problem of this class is, for instance, the problem of topology, to find a complete set of effectively calculable invariants of closed three-dimensional simplicial manifolds under homeomorphisms. This problem can be interpreted as a problem of elementary number theory in view of the fact that topological complexes are representable by matrices of incidence. In fact, as is well known, the property of a set of incidence matrices that it represent a closed three-dimensional manifold, and the property of two sets of incidence matrices that they represent homeomorphic complexes, can both be described in purely number-theoretic terms. If we enumerate, in a straightforward way, the sets of incidence matrices which represent closed three-dimensional manifolds, it will then be immediately provable that the problem under consideration (to find a complete set of effectively calculable invariants of closed three-dimensional manifolds) is equivalent to the problem, to find an effectively calculable function $f$ of positive integers, such that $f(m, n)$ is equal to 2 if and only if the $m$-th set of incidence matrices and the $n$-th set of incidence matrices in the enumeration represent homeomorphic complexes.

Other examples will readily occur to the reader.

---

[1] Presented to the American Mathematical Society, April 19, 1935.

[2] The selection of the particular positive integer 2 instead of some other is, of course, accidental and non-essential.

# ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

*By* A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers $\pi$, $e$, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

# AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

**1. Introduction.** There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function $f$ of $n$ positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer $n$ whether or not there exist positive integers $x, y, z$, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function $f$, such that $f(n)$ is equal to 2 if and only if there exist positive integers $x, y, z$, such that $x^n + y^n = z^n$. Clearly the condition that the function $f$ be effectively calculable is an essential part of the problem, since without it the problem becomes trivial.

Another example of a problem of this class is, for instance, the problem of topology, to find a complete set of effectively calculable invariants of closed three-dimensional simplicial manifolds under homeomorphisms. This problem can be interpreted as a problem of elementary number theory in view of the fact that topological complexes are representable by matrices of incidence. In fact, as is well known, the property of a set of incidence matrices that it represent a closed three-dimensional manifold, and the property of two sets of incidence matrices that they represent homeomorphic complexes, can both be described in purely number-theoretic terms. If we enumerate, in a straightforward way, the sets of incidence matrices which represent closed three-dimensional manifolds, it will then be immediately provable that the problem under consideration (to find a complete set of effectively calculable invariants of closed three-dimensional manifolds) is equivalent to the problem, to find an effectively calculable function $f$ of positive integers, such that $f(m, n)$ is equal to 2 if and only if the $m$-th set of incidence matrices and the $n$-th set of incidence matrices in the enumeration represent homeomorphic complexes.

Other examples will readily occur to the reader.

[1] Presented to the American Mathematical Society, April 19, 1935.

[2] The selection of the particular positive integer 2 instead of some other is, of course, accidental and non-essential.

# Turing Completeness

## and the Church-Turing thesis

# Weird sub-set of Elixir

A programming challenge!

# Weird sub-set of Elixir

Valid **terms** can be:

# Weird sub-set of Elixir

## Valid **terms** can be:

- Variable names such as `x`, `y`, or `my_variable`

# Weird sub-set of Elixir

## Valid **terms** can be:

- Variable names such as `x`, `y`, or `my_variable`
- Anonymous functions definitions like `fn x -> BODY end` where `BODY` is also a valid **term**.

# Weird sub-set of Elixir

## Valid **terms** can be:

- Variable names such as `x`, `y`, or `my_variable`
- Anonymous functions definitions like `fn x -> BODY end` where `BODY` is also a valid **term**.
- Application of functions, like `A.(B)` where both `A` and `B` are valid **terms**.

# Weird sub-set of Elixir

```
fn x -> x end
fn x -> x.(x) end
(fn x -> x end).(fn x -> x end)
fn _ -> fn x -> x end end
fn f -> fn x -> f.(x) end end
fn f -> fn x -> f.(f.(f.(f.(x)))) end end
```

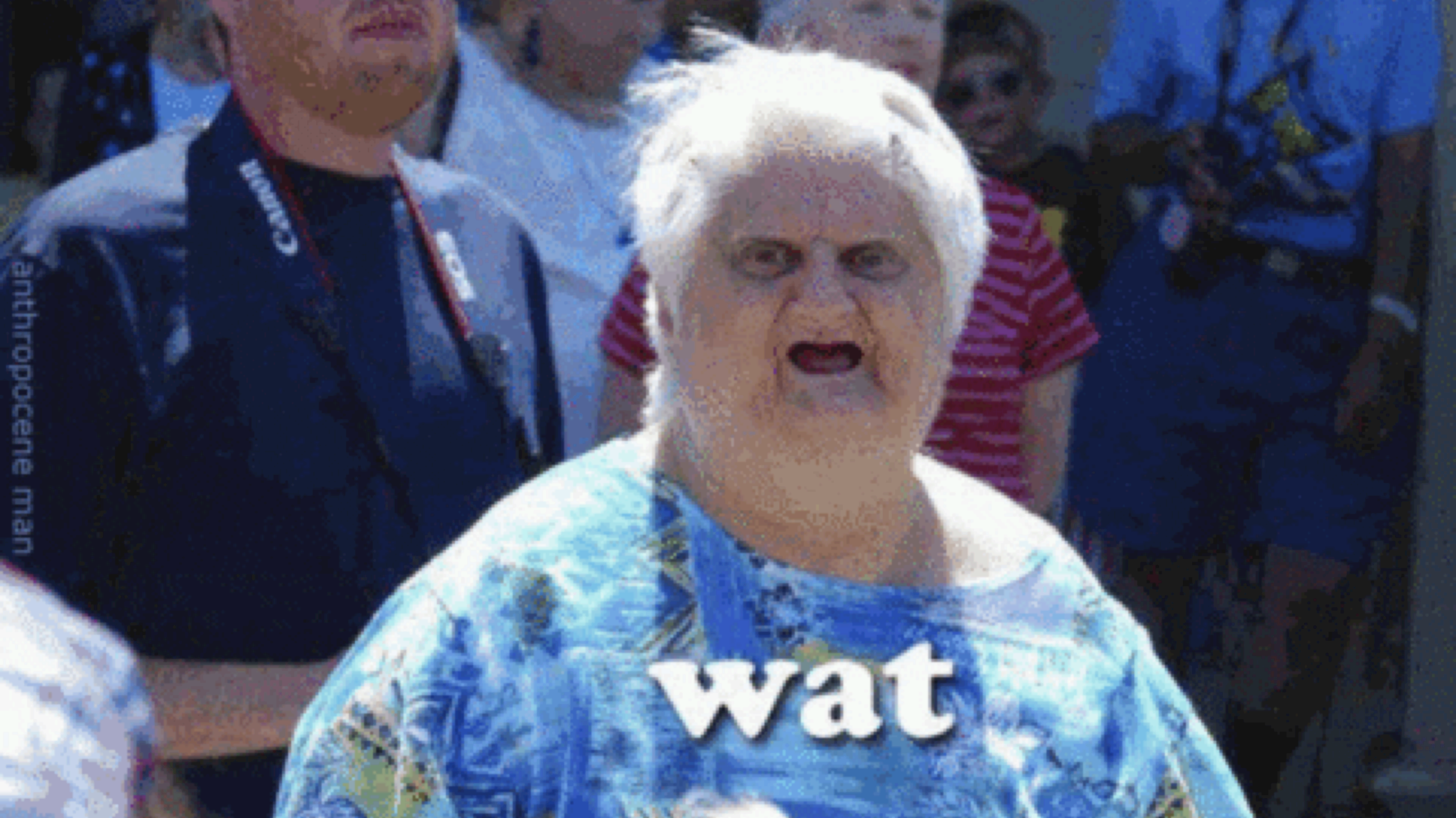
# Weird sub-set of Elixir

This is **Turing-Complete**

# Don't trust me?

Here is a factorial function.

```
(fn f -> (fn x -> x.(x) end).(fn x -> f.(
fn y -> x.(x).(y) end) end) end).(fn fact ->
fn n -> (fn b -> fn tf -> fn ff -> b.(tf).(ff).(b)
end end end).((fn n -> n.(fn _ -> fn _ -> fn f ->
f end end end).(fn t -> fn _ -> t end end) end).
(n)).(fn _ -> fn f -> fn x -> f.(x) end end end).
(fn _ -> (fn n -> fn m -> fn f -> fn x -> n.(m.(f))
.(x) end end end end).(n).(fact.((fn n -> fn f ->
fn x -> n.(fn g -> fn h -> h.(g.(f)) end end).
(fn _ -> x end).(fn u -> u end) end end end).(n)))
end) end end)
```


wat
anthropocene man

```
iex(5)> fact = (fn f -> (fn x -> x.(x) end).(fn x -> f.(fn y
-> x.(x).(y) end) end) end).(fn fact -> fn n -> (fn b -> fn
tf -> fn ff -> b.(tf).(ff).(b) end end end).((fn n -> n.(fn
_ -> fn _ -> fn f -> f end end end).(fn t -> fn _ -> t end
end) end).(n)).(fn _ -> fn f -> fn x -> f.(x) end end end).
(fn _ -> (fn n -> fn m -> fn f -> fn x -> n.(m.(f)).(x) end
end end end).(n).(fact.((fn n -> fn f -> fn x -> n.(fn g ->
fn h -> h.(g.(f)) end end).(fn _ -> x end).(fn u -> u end)
end end end).(n))) end) end end)
#Function<7.91303403/1 in :erl_eval.expr/5>
```

# Encoding and Decoding

```
5
|> number_to_lambda.()
|> fact.()
|> lambda_to_number.()
```

# Encoding and Decoding

```
iex(6)> 5 |>
...(6)> number_to_lambda.() |>
...(6)> fact.() |>
...(6)> lambda_to_number.()
120
```

# OK, What about λ-calculus?

# λ-calculus

# λ-calculus

- Formalism that defines computability

# λ-calculus

- Formalism that defines computability
- Based on simple functions that:

# λ-calculus

- Formalism that defines computability
- Based on simple functions that:
  - Are anonymous

# λ-calculus

- Formalism that defines computability
- Based on simple functions that:
  - Are anonymous
  - Are curried (1 argument function only)

# λ-calculus

- Formalism that defines computability
- Based on simple functions that:
  - Are anonymous
  - Are curried (1 argument function only)
- Defines a simple syntax for defining a **Lambda Term**

# λ-calculus syntax

| **Constructor** | **Lambda** |
| --- | --- |
| Variable | `x`, `y`, `my_var` |
| Abstraction | `λx. BODY` |
| Application | `A B` |

# Application is left associative

$$a\ b\ c = (a\ b)\ c$$

$$a\ b\ c \neq a\ (b\ c)$$

# Remember this?

```
fn x -> x end
fn x -> x.(x) end
(fn x -> x end).(fn x -> x end)
fn _ -> fn x -> x end end
fn f -> fn x -> f.(x) end end
fn f -> fn x -> f.(f.(f.(f.(x)))) end end
```

# In λ-calculus

```
λx. x
λx. x x
(λx. x) (λx. x)
λf. λx. x
λf. λx. f x
λf. λx. f f f f x
```

# More Examples

# More Examples

- `λa. λb. b a`

```
fn a -> fn b -> a.(b) end end
```

# More Examples

- λa. λb. b a

  ```
  fn a -> fn b -> a.(b) end end
  ```

- λa. λb. b b a

  ```
  fn a -> fn b -> b.(b).(a) end end
  ```

# More Examples

- λa. λb. b a

```
fn a -> fn b -> a.(b) end end
```

- λa. λb. b b a

```
fn a -> fn b -> b.(b).(a) end end
```

- λa. λb. b (b a)

```
fn a -> fn b ->  b.(b.(a)) end end
```

# More Examples

- λa. λb. b a

```
fn a -> fn b -> a.(b) end end
```

- λa. λb. b b a

```
fn a -> fn b -> b.(b).(a) end end
```

- λa. λb. b (b a)

```
fn a -> fn b ->  b.(b.(a)) end end
```

- λa. (λb. b) a)

```
fn a -> (fn b ->  b end).(a) end
```

# The simplest λ-term

The identity function

# The simplest λ-term

## The identity function

- λx. x

# The simplest λ-term

## The identity function

- `λx. x`
- `fn x -> x end`

# The simplest λ-term

```
iex(1)> id = fn x -> x end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(2)> id.(true)
true
```

# From λ to Factorial

# From λ to Factorial

```
def fact(n) do
  if n == 0 do
    1
  else
    n * fact(n-1)
  end
end
```

# From λ to Factorial

```
def fact(n) do
  if(
    n == 0,
    do: 1,
    else: n * (
      fact(
        n-1
  )))
end
```

# From λ to Factorial

```
def fact(n) do
  if(
    n == 0,
    do: 1,
    else: n * (
      fact(
        n-1
  )))
end
```

# From λ to Factorial

```
def fact(n) do
  if(
    n == 0,
    do: 1,
    else: n * (
      fact(
        n-1
  )))
end
```

# From λ to Factorial

```
def fact(n) do
  if(
    n == 0,
    do: 1,
    else: n * (
      fact(
        n-1
  )))
end
```

# From λ to Factorial

```
def fact(n) do
  if(
    n == 0,
    do: 1,
    else: n * (
      fact(
        n-1
  )))
end
```

# From λ to Factorial

```
def fact(n) do
  if(
    n == 0,
    do: 1,
    else: n * (
      fact(
        n-1
  )))
end
```

# From λ to Factorial

```
def fact(n) do
  if(
    n == 0,
    do: 1,
    else: n * (
      fact(
        n-1
  )))
end
```

# So we need

- □ Encoding for Natural Numbers
- □ Encoding for Booleans
- □ Function to check if number is zero
- □ If/Then/Else
- □ Multiplication
- □ Predecessor
- □ Recursion

# Encoding Booleans

# That is: encode **True** and **False**

**P.S.: There are infinite ways of doing this**

# What are booleans used for?

# Branching

Pick one of two paths

It must be a function that will receive another function, at least

λ?. ???

# Church Booleans

λthen. λelse. ???

True: λthen. λelse. then

False: λthen. λelse. else

# In Elixir

```
# True
fn then_path -> fn _ -> then_path end end

# False
fn _ -> fn false_path -> false_path end end
```

```
iex(3)> true! = fn t -> fn _ -> t end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(4)> false! = fn _ -> fn f -> f end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(5)> true!.("This if true").("This if false")
"This if true"
iex(6)> false!.("This if true").("This if false")
"This if false"
```

# Decoding Booleans

Need a way to check the result

# The output on iex is not helpful

```
#Function<7.91303403/1 in :erl_eval.expr/5>
```

# Let's cheat

We can apply non-lambda terms to our lambda term

# Let's cheat

We can apply non-lambda terms to our lambda term

```
iex(7)> lambda_to_bool = fn b -> b.(true).(false) end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(8)> lambda_to_bool.(true!)
true
iex(9)> lambda_to_bool.(false!)
false
```

# Operations on Booleans

# Not Function

# Not Function

| **a** | **not a** |
| --- | --- |
| `true` | `false` |
| `false` | `true` |

# Not Function

λa. ???

# Not Function

```
λa. a FALSE TRUE
```

# Not Function

```
fn a ->
  a.(false!).(true!)
end
```

# Not Function

```
iex(10)> not! = fn a -> a.(false!).(true!) end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(11)> true! |> not!.() |> lambda_to_bool.()
false
iex(12)> false! |> not!.() |> lambda_to_bool.()
true
```

# And Function

# And Function

| a | b | and a b |
|---|---|---|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

# And Function

```
λa. λb. ???
```

# And Function

```
λa. λb. a ??? ???
```

# And Function

```
λa. λb. a ??? FALSE
```

# And Function

```
λa. λb. a b FALSE
```

# And Function

```
fn a -> fn b ->
  a.(b).(false!)
end end
```

# And Function

```
iex(13)> and! = fn a -> fn b -> a.(b).(false!) end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(14)> and!.(true!).(true!) |> lambda_to_bool.()
true
iex(15)> and!.(true!).(false!) |> lambda_to_bool.()
false
iex(16)> and!.(false!).(true!) |> lambda_to_bool.()
false
iex(17)> and!.(false!).(false!) |> lambda_to_bool.()
false
```

# Or Function

# Or Function

- This is similar to and function, but short circuits on true instead of false

# Or Function

- This is similar to and function, but short circuits on true instead of false
  - `λa. λb. a TRUE b`

# Or Function

- This is similar to and function, but short circuits on true instead of false
  - `λa. λb. a TRUE b`
- And it is proven that if you have `and` and `not` you can create all other logic gates

# Or Function

- This is similar to and function, but short circuits on true instead of false
  - `λa. λb. a TRUE b`
- And it is proven that if you have `and` and `not` you can create all other logic gates
  - You actually just need `nand` which is a combination of `and` and `not`.

# What we have so far

- □ Encoding for Natural Numbers
- ☑ Encoding for Booleans
- □ Function to check if number is zero
- □ If/Then/Else
- □ Multiplication
- □ Predecessor
- □ Recursion

# Encoding Natural Numbers

# That is: encode **0**, **1**, **2**, ...

**P.S.: There are also infinite ways of doing this**

# What natural numbers are used for?

# Counting things

# Church Numerals

Count the number of times a function is applied to a given input

```
N => λf. λx. F_APPLIED_TO_X_N_TIMES
```

| Number | Encoding |
| --- | --- |
| 0 | `λf. λx. x` |
| 1 | `λf. λx. f x` |
| 2 | `λf. λx. f (f x)` |
| 3 | `λf. λx. f (f (f x))` |
| 4 | `λf. λx. f (f (f (f x)))` |

# Constructing Natural Numbers

# Constructing Natural Numbers

- We need zero

# Constructing Natural Numbers

- We need zero
- And a way to get N+1 given N (successor)

# Zero

```
λf. λx. x
```

# Zero

```
fn _f -> fn x -> x end end
```

# Successor Function

λn. ???

# Successor function

```
λn. λf. λx. ???
```

# Successor function

```
λn. λf. λx. ??? (n f x)
```

# Successor function

```
λn. λf. λx. f (n f x)
```

# Successor Function

```
fn n -> fn f -> fn x ->
  f.(
    n.(f).(x)
  )
end end end
```

# Constructing Church Numerals

```
iex(18)> zero = fn _f -> fn x -> x end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(19)> succ = fn n -> fn f -> fn x -> f.(n.(f).(x)) end end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(20)> one = succ.(zero)
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(21)> two = succ.(succ.(zero))
#Function<7.91303403/1 in :erl_eval.expr/5>
```

# The output on iex is not helpful (again)

```
#Function<7.91303403/1 in :erl_eval.expr/5>
```

# Encoding and Decoding Church Numerals

```
# Encode
fn n -> n.(&(&1 + 1)).(0) end

# Decode
fn
  0 -> zero
  n -> Enum.reduce(1..n, zero, fn _, x -> succ.(x) end)
end
```

```
iex(22)> lambda_to_number = fn n -> n.(&(&1 + 1)).(0) end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(23)> number_to_lambda = fn
...(23)>   0 -> zero
...(23)>   n -> Enum.reduce(1..n, zero, fn _, x -> succ.(x) end)
...(23)> end
#Function<7.91303403/1 in :erl_eval.expr/5>
```

```
iex(24)> lambda_to_number.(zero)
0
iex(25)> lambda_to_number.(one)
1
iex(26)> lambda_to_number.(two)
2
iex(27)> lambda_to_number.(succ.(two))
3
iex(28)> 10 |> number_to_lambda.() |> succ.() |> lambda_to_number.()
11
```

# What we have so far

- ☑ Encoding for Natural Numbers
- ☑ Encoding for Booleans
- □ Function to check if number is zero
- □ If/Then/Else
- □ Multiplication
- □ Predecessor
- □ Recursion

# Is Zero?

# Is Zero?

λn. ???

# Is Zero?

```
λn. n ?F? ?X?
```

# Is Zero?

```
λn. n ?F? TRUE
```

# Is Zero?

```
λn. n (λx. FALSE) TRUE
```

# Is Zero?

```
fn n ->
  n.(fn _ -> false! end).(true!)
end
```

```
iex(29)> is_zero = fn n -> n.(fn _ -> false! end).(true!) end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(30)> zero |> is_zero.() |> lambda_to_bool.()
true
iex(31)> one |> is_zero.() |> lambda_to_bool.()
false
iex(32)> two |> is_zero.() |> lambda_to_bool.()
false
```

# What we have so far

- ☑ Encoding for Natural Numbers
- ☑ Encoding for Booleans
- ☑ Function to check if number is zero
- ☐ If/Then/Else
- ☐ Multiplication
- ☐ Predecessor
- ☐ Recursion

# If Then Else

# If Then Else

- Isn't it just applying booleans to the branches?

# If Then Else

- Isn't it just applying booleans to the branches?
- No, because Elixir is **eagerly evaluated**

# If Then Else

- Isn't it just applying booleans to the branches?
- No, because Elixir is **eagerly evaluated**
- So we'll need to do one trick

```
λb. λthen. λelse. b then else ANY_VALUE
```

```
# ifte
fn b -> fn t -> fn e ->
  b.(t).(e).(b)
end end end

# Usage

ifte.(fn _ -> ... end).(fn _ -> ... end)
```

```
iex(33)> ifte = fn b -> fn t -> fn e -> b.(t).(e).(b) end end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(34)> ifte.(true!).(fn _ -> true end).(fn _ -> false end)
true
iex(35)> ifte.(false!).(fn _ -> true end).(fn _ -> false end)
false
```

# What we have so far

- ☑ Encoding for Natural Numbers
- ☑ Encoding for Booleans
- ☑ Function to check if number is zero
- ☑ If/Then/Else
- ☐ Multiplication
- ☐ Predecessor
- ☐ Recursion

# Multiplication

# Multiplication

```
λa. λb. ???
```

# Multiplication

```
λa. λb. λf. λx. ???
```

# Multiplication

```
λa. λb. λf. λx. ??? (a f) ???
```

# Multiplication

```
λa. λb. λf. λx. b (a f) ???
```

# Multiplication

```
λa. λb. λf. λx. b (a f) x
```

```
fn a -> fn b -> fn f -> fn x ->
  b.(
    a.(f)
  ).(x)
end end end end
```

```
iex(36)> mul = fn a -> fn b -> fn f -> fn x ->
...(36)>   b.(a.(f)).(x)
...(36)> end end end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(37)> mul.
...(37)>   (number_to_lambda.(5)).
...(37)>   (number_to_lambda.(10)) |> lambda_to_number.()
50
```

# What we have so far

- ☑ Encoding for Natural Numbers
- ☑ Encoding for Booleans
- ☑ Function to check if number is zero
- ☑ If/Then/Else
- ☑ Multiplication
- ☐ Predecessor
- ☐ Recursion

# Predecessor Function

Should be simple, right?

Nope

I'll left as an exercise to the reader
- All book writers

# Predecessor Function

```
λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)
```

```
iex(38)> pred = fn n -> fn f -> fn x ->
...(38)>   n.(fn g -> fn h -> h.(g.(f)) end end).(fn _ -> x end).(fn u -> u end)
...(38)> end end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(39)> 10 |> number_to_lambda.() |> pred.() |> lambda_to_number.()
9
```

# What we have so far

- ☑ Encoding for Natural Numbers
- ☑ Encoding for Booleans
- ☑ Function to check if number is zero
- ☑ If/Then/Else
- ☑ Multiplication
- ☑ Predecessor
- ☐ Recursion

# Recursion

# The problem with recursion

# The problem with recursion

- All functions are anonymous

# The problem with recursion

- All functions are anonymous
- Therefore, we cannot refer a function to itself by name

# The problem with recursion

- All functions are anonymous
- Therefore, we cannot refer a function to itself by name
- Solution: **Abstract yourself**

# Abstract Yourself

# Abstract Yourself

- λx. ?CALL_MY_SELF?

# Abstract Yourself

- `λx. ?CALL_MY_SELF?`
- `λmyself. λx. myself x`

```
λfact. λn.
  IFTE
    (IS_ZERO n)
    (λx. ONE)
    (λx. (MUL n (fact (PRED n))))
```

```
fn fact -> fn n ->
  ifte.(is_zero.(n)).(
    fn _ -> one end
  ).(
    fn _ -> mul.(n).(fact.(pred.(n))) end
  )
end end
```

```
iex(40)> abs_fact = fn fact -> fn n ->
...(40)>   ifte.(is_zero.(n)).(
...(40)>     fn _ -> one end
...(40)>   ).(
...(40)>     fn _ -> mul.(n).(fact.(pred.(n))) end
...(40)>   )
...(40)> end end
#Function<7.91303403/1 in :erl_eval.expr/5>
```

# How do we use that?

# How do we use that?

- We need a "non abstracted" factorial

# How do we use that?

- We need a "non abstracted" factorial
- To build it, we call the abstracted one with something

# How do we use that?

- We need a "non abstracted" factorial
- To build it, we call the abstracted one with something
- This something is the "non abstracted" factorial

# How do we use that?

- We need a "non abstracted" factorial
- To build it, we call the abstracted one with something
- This something is the "non abstracted" factorial
- Circular dependency?

# First solution: Fake it!

# First solution: Fake it!

- When n is 0, factorial is not called

# First solution: Fake it!

- When n is 0, factorial is not called
- Plug whatever function we want

# First solution: Fake it!

```
iex(41)> 0 |> number_to_lambda.() |> abs_fact.(id).() |> lambda_to_number.()
1
iex(42)> 1 |> number_to_lambda.() |> abs_fact.(id).() |> lambda_to_number.()
0
```

# Why it does not work for 1?

# Why it does not work for 1?

- When n is 1, the function calls factorial again

# Why it does not work for 1?

- When n is 1, the function calls factorial again
- But factorial is not factorial, is id

# Why it does not work for 1?

- When n is 1, the function calls factorial again
- But factorial is not factorial, is id
- Id of n-1 is 0, then the multiplication result is 0

# Solution: Fake it one more time

```
iex(43)> 1 |> number_to_lambda.() |> abs_fact.(abs_fact.(id)).() |> lambda_to_number.()
1
iex(44)> 2 |> number_to_lambda.() |> abs_fact.(abs_fact.(id)).() |> lambda_to_number.()
0
```

# We can keep faking it

```
iex(45)> 3 |>
...(45)> number_to_lambda.() |>
...(45)> abs_fact.(abs_fact.(abs_fact.(abs_fact.(id)))).() |>
...(45)> lambda_to_number.()
6
iex(46)> 4 |>
...(46)> number_to_lambda.() |>
...(46)> abs_fact.(abs_fact.(abs_fact.(abs_fact.(id)))).() |>
...(46)> lambda_to_number.()
0
```

# What we need to do?

# What we need to do?

- How many times we need to apply the abstract function?

# What we need to do?

- How many times we need to apply the abstract function?
- One for every recursion

# What we need to do?

- How many times we need to apply the abstract function?
- One for every recursion
- We know that factorial of N is going to take N recursion

```
λn. abs_fact (?ABS_FACT_APPLIED_N_TIMES?) n
```

```
λn. abs_fact (n abs_fact id) n
```

```
fn n ->
  abs_fact.(n.(abs_fact).(id)).(n)
end
```

```
iex(47)> faux_fact = fn n -> abs_fact.(n.(abs_fact).(id)).(n) end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(48)> 3 |> number_to_lambda.() |> faux_fact.() |> lambda_to_number.()
6
iex(49)> 4 |> number_to_lambda.() |> faux_fact.() |> lambda_to_number.()
24
iex(50)> 5 |> number_to_lambda.() |> faux_fact.() |> lambda_to_number.()
120
```

Is that general recursion?

# What if we don't know how many recursions will be necessary?

# What we have and what we need?

# What we have and what we need?

- We have an abstracted recursive function (`abs_fact`)

# What we have and what we need?

- We have an abstracted recursive function (`abs_fact`)
- We need to be able to generate something like this `abs_fact (abs_fact (abs_fact ...))`

# What we have and what we need?

- We have an abstracted recursive function (`abs_fact`)
- We need to be able to generate something like this `abs_fact (abs_fact (abs_fact ...))`
- On demand / infinitely

# Fixed-point combinators

# Fixed-point combinators

- Must have the property:
  `fix f = f (fix f)`

# Fixed-point combinators

- Must have the property:
  `fix f = f (fix f)`
- Which if you expand once, you have:
  `= f (f (fix f))`

# Fixed-point combinators

- Must have the property:
  `fix f = f (fix f)`
- Which if you expand once, you have:
  `= f (f (fix f))`
- And if you keep expanding:
  `= f (f (f (... (f (fix f)) ...)))`

# Fixed-point combinators

- Must have the property:
  `fix f = f (fix f)`
- Which if you expand once, you have:
  `= f (f (fix f))`
- And if you keep expanding:
  `= f (f (f (... (f (fix f)) ...)))`
- Good news: there is also infinite of these

# Y-Combinator

# Y-Combinator

- It's not just an accelerator

# Y-Combinator

- It's not just an accelerator
- It's a fixed-point combinator for lambda calculus

# Y-Combinator

- It's not just an accelerator
- It's a fixed-point combinator for lambda calculus
- Discovered by **Haskell B. Curry**

# Y-Combinator

- It's not just an accelerator
- It's a fixed-point combinator for lambda calculus
- Discovered by **Haskell B. Curry**
- Defined as: `Y = λf (λx. f (x x)) (λx. f (x x)))`

# In Elixir

```
# Can't I just do
y_combinator = fn ... end

# And apply
fact = y_combinator.(abs_fact)

# And use
fact.(...)
```

# Elixir is Eagerly Evaluated

# Elixir is Eagerly Evaluated

- It evaluates things before passing as arguments

# Elixir is Eagerly Evaluated

- It evaluates things before passing as arguments
- That's why our `ifte` had then and else as functions

# Elixir is Eagerly Evaluated

- It evaluates things before passing as arguments
- That's why our `ifte` had then and else as functions
- That's why macros are used if you want to implement your if

# Elixir is Eagerly Evaluated

- It evaluates things before passing as arguments
- That's why our `ifte` had then and else as functions
- That's why macros are used if you want to implement your if
- That's why the Y-Combinator does not work

# Z-Combinator

# Z-Combinator

- Another fixed-point combinator

# Z-Combinator

- Another fixed-point combinator
- Works in eagerly evaluated environments

# Z-Combinator

- Another fixed-point combinator
- Works in eagerly evaluated environments
- Trick is similar to what we've done in `ifte`

# Z-Combinator

- Another fixed-point combinator
- Works in eagerly evaluated environments
- Trick is similar to what we've done in `ifte`
- Defined as: `Z = λf. (λx. x x) (λx. f (λy. x x y))`

```
iex(51)> z_combinator = fn f ->
...(51)>   (fn x -> x.(x) end).(
...(51)>     fn x -> f.(fn y -> x.(x).(y) end) end
...(51)>   )
...(51)> end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(52)> fact = z_combinator.(abs_fact)
#Function<7.91303403/1 in :erl_eval.expr/5>
```

```
iex(53)> 3 |> number_to_lambda.() |> fact.() |> lambda_to_number.()
6
iex(54)> 4 |> number_to_lambda.() |> fact.() |> lambda_to_number.()
24
iex(55)> 5 |> number_to_lambda.() |> fact.() |> lambda_to_number.()
120
```

# Thanks Folks!

github.com/bamorim/elixir-lambda-talk