Playing with Lambda Calculus

Bernardo Amorim

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 π, 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

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 ≠ a (b c)

$$\lambda x.\ \lambda y.\ y\ x \neq \lambda x.\ (\lambda y.\ y)\ x$$

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

Lambda Calculus in Elixir

Programming Challenge

Weird sub-set of Elixir

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.(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
```

Remember this?

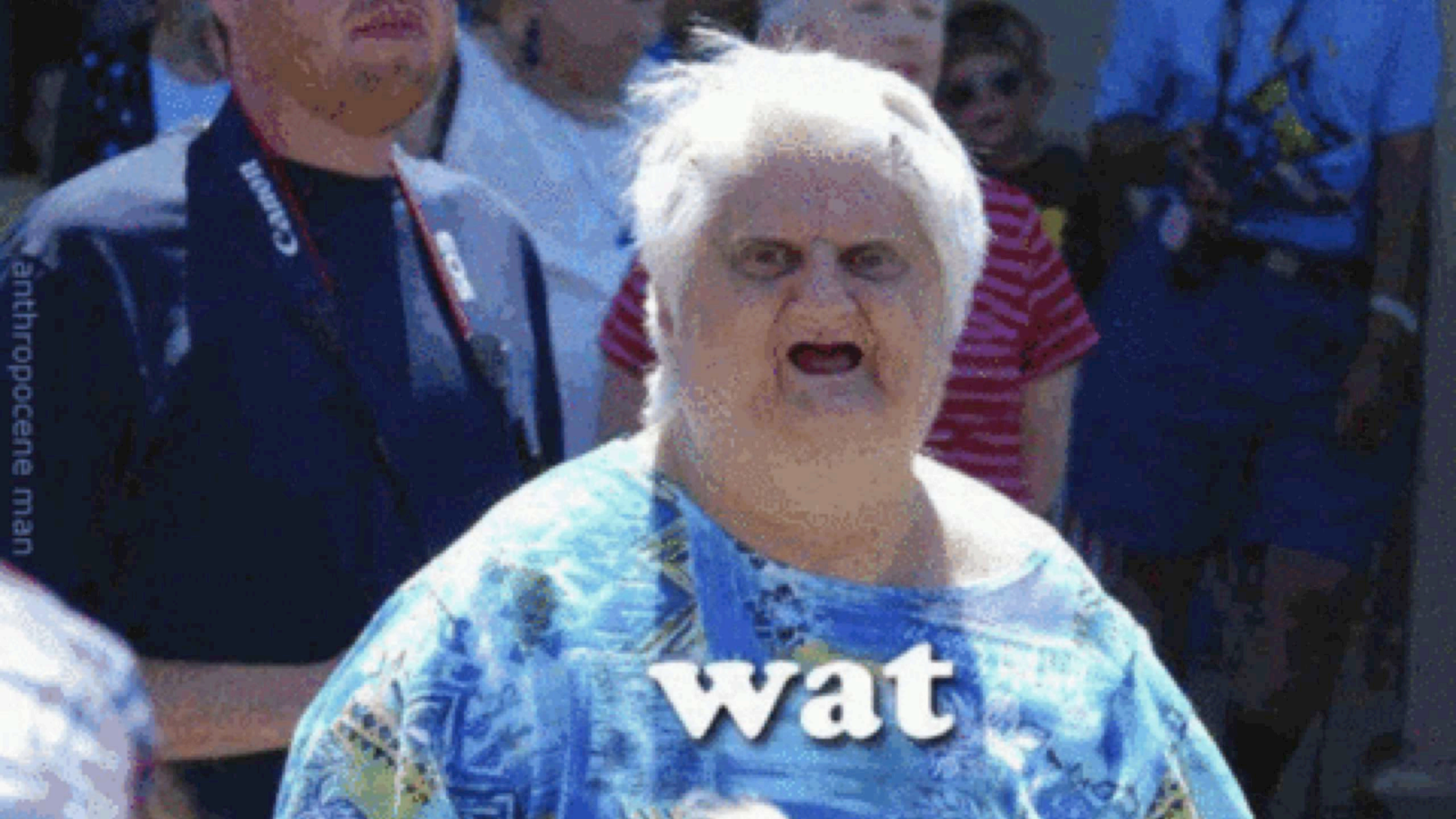
```
λx. x
λx. x x
λx. x x x
(λx. x) (λx. x)
λf. λx. x
λf. λx. f x
λf. λx. f (f (f (f x)))
```

Weird sub-set of Elixir

This is **Turing-Complete**

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
```

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
```

Boolean Encoding in Lambda Terms

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

$\lambda ?.\ ???$

λthen. λelse. ???

True: `λthen. λelse. then`

False: `λthen. λelse. else`

Church Booleans

In Elixir

```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"
```

```
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

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

```
encoded_boolean.(true).(false)
```

```
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

Negation Function

Negation Function

a	**not a**
true	false
false	true

Negation Function

`λa. ???`

Negation Function

```
λa. a ?WHEN_TRUE? ?WHEN_FALSE?
```

Negation Function

```
λa. a FALSE TRUE
```

Negation Function

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

Negation 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
```

Other Logic Gates

NAND Logic

With `not` and `and` you can implement all other gates

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. ???)`

Apply `f` to x N+1 times

Applying N times

`n f x`

Successor function

```
λn. (λf. λx. ??? (n f x))
```

Successor function

```
λn. (λf. λx. f (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
```

```
ex(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>
```

```
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>
```

Elixir Numbers ↔ Church Numerals

Elixir Numbers ↔ Church Numerals

```elixir
lambda_to_number = fn n ->
  n. # Do N times
    (&(&1 + 1)). # Adds 1
    (0) # Start with 0
end

number_to_lambda = fn n ->
  0..n |> Enum.drop(1) |> Enum.reduce(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 n ->
...(23)>   0..n |> Enum.drop(1) |> Enum.reduce(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
```

Addition

Addition

`A + 0`	`A`
`A + 1`	`SUCC A`
`A + 2`	`SUCC (SUCC A)`
`A + 3`	`SUCC (SUCC (SUCC A))`
`A + B`	SUCC applied B times to A

Addition

```
λa. λb. ?SUCC_APPLIED_B_TIMES_TO_A?
```

Addition

```
λa. λb. b ?F? a
```

Addition

```
λa. λb. b SUCC a
```

Addition

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

```
iex(29)> add = fn a -> fn b -> b.(succ).(a) end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(30)> zero |> add.(one).() |> lambda_to_number.()
1
iex(31)> one |> add.(one).() |> lambda_to_number.()
2
iex(32)> two |> add.(two).() |> lambda_to_number.()
4
```

Multiplication

Multiplication

`A * 0`	`0`
`A * 1`	`0 + A`
`A * 2`	`0 + A + A`
`A * 3`	`0 + A + A + A`
`A * B`	A added to `0` B times

Multiplication

```
λa. λb. ?A_ADDED_TO_ZERO_B_TIMES?
```

Multiplication

```
λa. λb. b ?ADD_A? ZERO
```

Multiplication

```
λa. λb. b (λx. ADD a x) ZERO
```

Multiplication

```
λa. λb. b (ADD a) ZERO
```

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

```
iex(33)> mul = fn a -> fn b -> b.(add.(a)).(zero) end end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(34)> mul.
...(34)>   (number_to_lambda.(5)).
...(34)>   (number_to_lambda.(10)) |> lambda_to_number.()
50
```

What's next?

Predecessor Function

Predecessor Function

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

Recursion

Fixed Point Combinators

That's all, folks.

(for now)

github.com/bamorim/elixir-lambda-talk