# Lambda Calculus and its Impact on Computer Science

Joris Klaasse Bos
Zaanlands Lyceum

6th November 2020

## Preface

The reason I chose to write about this subject, is that it combines two things I enjoy: abstract maths and computer science. I have been programming for about five to six years now. I mainly enjoy low-level programming, so—naturally—C is my most used language and I am most familiar with a simple procedural paradigm. Such a paradigm is, however, not always very easy to use when working with very large and complex systems. I, as many, started out with object-oriented programming (OOP), but I did not like that very much. Therefore, I have been exploring alternative paradigms, including data-oriented programming and functional programming. I am quite familiar with data-oriented programming and the Rust programming language by now, but functional programming isn't something I've ever really got into yet. I did find out about lambda calculus and combinatory logic, which intrigued me, but I haven't got into it beyond a basic level of understanding. That is why I decided to research it for this work.

# Contents

**Preface** **2**

**1 Introduction** **4**

**2 Introduction to lambda calculus** **4**
2.1 A short history . . . 4
2.2 The syntax . . . 5
2.3 Combinatory logic . . . 7
2.3.1 Identity . . . 7
2.3.2 The omega combinator . . . 7
2.3.3 The constant combinator . . . 8
2.3.4 The kite . . . 8
2.3.5 The flip combinator . . . 9
2.3.6 The composition combinator . . . 9
2.3.7 The thrush . . . 10
2.3.8 The starling . . . 10
2.3.9 $S$ and $K$ . . . 10
2.3.10 To Mock a Mockingbird . . . 10

**3 Using lambda calculus for computation** **12**
3.1 Church encoding . . . 12
3.1.1 Church booleans . . . 12
3.1.2 Church numerals . . . 16
3.1.3 Merging of boolean operations and arithmetic . . . 19
3.1.4 Data structures . . . 20
3.2 Recursion . . . 20

**4 Functional programming (lambda calculus applied)** **20**
4.1 Why do we even care? . . . 20
4.2 Lambda functions . . . 20
4.3 Laziness . . . 20
4.4 Types . . . 20
4.5 Monads . . . 20
4.6 Haskell . . . 20
4.7 Comparison to with other paradigms . . . 20
4.7.1 Declarative vs imperative . . . 20
4.7.2 Usefulness vs conceptual purity . . . 20
4.7.3 Meta programming . . . 20

**5 Functional programming in other paradigms** **20**
5.1 Lambda functions . . . 20
5.2 Iterators . . . 20

**6 Possibilities for the future** **20**
6.1 Conceptual *and* useful . . . 20

**Afterword** **20**

**References** **21**

# 1 Introduction

With the decline of OOP, many other paradigms are gaining in popularity. One increasingly popular paradigm is functional programming. Functional programming is fundamentally based on lambda calculus and it has been seeping into other paradigms and into mainstream languages. Most of the popular languages now implement lambda functions and have ways to write in a more declarative style of programming. In this work I will look at all the ways lambda calculus has influenced computer science and how it may do so in the future.

# 2 Introduction to lambda calculus

Lambda calculus is, as its name suggests, a calculus. A calculus is a system of manipulating symbols, which by themselves don't have any semantic meaning, in a way that is somehow meaningful. We all know algebra. Algebra itself doesn't have an innate meaning, but we can use it to represent and solve real world problems. Algebra, however, is limited. Not every problem can be represented in algebra. There are many branches of mathematics that use different systems. One example would be formal logic, which is used for logical operations on booleans. Another such example is lambda calculus.

## 2.1 A short history

People always trusted mathematics to be true and relied on it heavily. If something was proven true with mathematical logic, then that must be true. However, starting from the late 19th century, people ran into paradoxes. People made a distinction between reasoning that is rigorous and reasoning that isn't—reasoning that is logical, and reasoning that is psychological. The fact that mathematical logic, which people looked at for rigorousness, is infested with paradoxes and self-reference was very troubling for people at the time.

The concept of mathematics and mathematical logic wasn't well defined, so people started to think about the formalisation of mathematical logic to try to solve these issues. People wanted a system that would encapsulate all of mathematical logic. Preferably this system would be simple, clean and intuïtive.

Throughout the late 19th and early 20th centuries, people started formally defining and redefining different aspects of mathematics. Frege [4] wrote about propositional calculus and functions as graphs, and in doing so reëvaluated the concept of functions and was already using concepts like Currying functions (more on this in section 2.2) without really giving thought to it. Peano [6] invented the Peano axioms and Peano arithmetic as a way of defining natural numbers. He was not the first to attempt defining natural numbers, but he was the most successful. Schönfinkel [8] invented combinatory logic, which was later rediscovered and improved on by Curry [3], as a way to remove the need for quantified variables in logic.

One major attempt to define all of mathematics was done by Russell and Whitehead [7]. They wrote a book that would become well know in all of mathematics and logic. This book is called *Principia Mathematica*. They did, however, run into a few problems, which arose from self-reference. To solve these problems that this paradoxical self-reference brought with it, they invented an elaborate system, the theory of types, to circumvent/eliminate it. It was a very carefully crafted bastion against self-reference ever coming up in their system, which was not very simple, clean or intuïtive.

People praised *PM* as they thought they had finally done it; they had formalised all of mathematical logic, they had realised the dream of grounding all of mathematics in logic. But in Vienna, Gödel was sceptical of this book. He started seeing some cracks, he felt that there was something wrong aboout this attempt. Gödel felt that self-reference was a fundamental part of mathematical logic. Then he went out and actually proved that there is no consistent system

of axioms whose theorems can be listed by an effective procedure that is capable of proving all truths about arithmetic of natural numbers[1] [5], meaning that such a system is either inconsistent or incomplete[2], greatly disturbing many mathematicians and upending mathematics as they knew it.

During this time, in this environment of the formalisation of mathematical logic, Church [1] invented the lambda calculus. Lambda calculus is a very simple and minimalistic system of substitution. A little while later, Turing [11, 12] invented Turing machines. Turing machines are conceptual mathematical machines that function based on state—they were state machines. These could perform all kinds of mathematical and logical computations. He was not the first to invent computers, but he was the first to work them out as well as he did (and, as you probably know, he built one which he cracked the German enigma code with).

There was this problem that has a few different names. It is often known as the *halting problem* or the *Entscheidungsproblem*, which is German for *decision problem*. The halting problem and decision problem aren't exactly synonyms, but they come down to the same thing. Basically, it asks wether it is possible to know via an algorithm wether a computation will complete execution or result in an infinite loop. In 1936, Turing [11, 12] spent a long time proving, using his Turing machines, that this isn't possible, but it didn't get published until early 1937. Also in 1936, Church [2] proved the same thing using lambda calculus and happended to publish it before Turing did. When Turing finally got around to publishing his proof, he found out that he was beaten to it by Church. He wasn't to pleased. What is interesting, though, is that lambda calculus and Turing machines take two completely different approaches. Turing machines funtion entirely on state, while lambda calculus is completely stateless (we'll look at this later). Turing thought this was interesting too, so he researched lambda calculus and how it relates to his Turing machines, and proved that they are formally equivalent [10].

Why do I tell you all this? Well, your main takeaway should be that even though lambda calculus is a very simple system, which at first glance might not seem to be very semantic or seem to have any real world implications, it actually is Turing complete. Lambda calculus and Turing machines take wildly different approaches: one state based, the other stateless. Another difference is that Turing machines can be physically built. We can, however, use lambda calculus on these Turing machines *and* simulate Turing machines with lambda calculus, which is a fundamental part to the thesis of this work. We will look at lambda calculus and how the work of all the previously mentioned mathematicians, and many more, can be applied in lambda calculus to get a useful system.

## 2.2 The syntax

Lambda calculus is all about first-class higher-order pure (anonymous)[3] unary functions. Such a function takes a single input, and returns a single expression that is only dependent on the input, so it doesn't have any outside state. Such a function can take and return any expression, which in lambda calculus is always a function. A simple function definition in lambda calculus looks as follows:

$$\lambda a.a$$

The lambda signifies a function. Everything following it will be part of that function's definition. The $a$ before the . is the name of the argument. There is only one, because, as I said before, all functions in lambda calculus are unary. Everything following the . is part of the function body, which is the return expression. The funcion above is the identity function in lambda calculus; it just returns the input. This is the equivalent of multiplying by one, or

[1] This is a definition of the first incompleteness theorem I got from Wikipedia [14]

[2] Incompleteness means that there are things that are true, but are not provable.

[3] The core lambda calculus has no way of naming functions.

defining a function like $f(x) = x$, or multiplying a vector with the identity matrix; it does nothing.

But how do we use this function? Well, just like defining a function, it is quite simple. If you want to apply this function to a symbol, you just put it in front of the symbol. Something like this:

$$(\lambda a.a)x$$

Which evaluates to $x$, because you remove the $x$ and then replace all the $a$'s in the function body with $x$ and then remove the function signifier and argument list. It all comes down to a simple process of substitution.

In this case you need parentheses around the function, otherwise $x$ would be considered part of the function's body, which it isn't. It's also important to note that lambda calculus is left-associative, that is, it evaluates an expression from left to right. This means that the function on the far left of an expression gets invoked first.

I have now basically explained the entire lambda calculus, it is really that simple. I have explained abstraction (functions), application (applying functions), and grouping (parentheses), which is basically all we need. You can also give names to expressions. We could name our identity function $I$ as follows:

$$I := \lambda a.a$$

But this isn't really part of the core lambda calculus anymore, just some syntactic sugar. This way, instead of constantly having to write $\lambda a.a$, we can just write $I$. So instead of writing:

$$(\lambda a.a)x = x$$

We could use our previous definition of $I$ and write:

$$Ix = x$$

We have now covered identifiers too.

But if this is all there is, how can this possibly be Turing complete? How do we do boolean logic, or algebra? How can we do things with only unary functions? What are $a$ and $x$ supposed to represent? If there is no concept of value, how do we even use this meaningfully? Well, the key is this: a function can return any expression (remember?), which is always a function[4], not just a single symbol. We can start combinating these simple functions into more complex functions. Let's say that we wanted to have a function that takes two arguments, and then applies the first argument to the second one. You are probably asking yourself a few questions. For example, what does it mean for one argument to be applied to another? Well, as I said, everything is a function. But the biggest question you are probably asking yourself is: how can you have a function that takes two arguments?

We actually can't, but what we can do is to have a function that takes one argument and returns another function that takes one argument. We can define that function as follows:

$$\lambda a.\lambda b.ab$$

We currently have a function definition inside the body of another function. If we now apply this function to a symbol like $x$, we get this:

$$(\lambda a.\lambda b.ab)x = \lambda b.xb$$

We get a new function that takes an argument and applies $x$ to it. If we now apply this function to a symbol like $y$, we get this:

$$(\lambda b.xb)y = xy$$

[4]Everything is.

Alternatively, we could write it all on one line:

$$(\lambda a.\lambda b.ab)xy = (\lambda b.xb)y = xy$$

$xy$ in this case is what we would call the *$\beta$-normal form* of the preceding expressions. That just means that it is in the simpelest form and isn't able to be evaluated any further. Reducing a lambda expression to the $\beta$-normal form is called *$\beta$-reduction*.

You can start to see how we can combine unary functions to create more complex functions[5]. In this example we used two nested unary functions to get the same result you would with a binary function. Such a nested function is often called a *Curry'd function*. You might think to yourself that having this many nested functions can be quite convoluted and not very readable, and you're quite right. That's why people often use a shorthand notation. They would basically write it as if it is a single binary function (as with any n-ary function). They would write the example function above as:

$$\lambda ab.ab$$

Do keep in mind, that even though this looks and, for the most part, acts as if it is a single binary function, it really isn't. It still is a Curry'd function that feeds in the arguments one by one, but this way the expression becomes more readable and easier to think about conceptually. We will use this notation from now on.

Congratulations, you now know the very basics of lambda calculus. You may still not see how this is Turing complete or how this can be useful and meaningful. You might also already see some of the intrigues of lambda calculus; how simple it is, how it doesn't have a concept of value or data, how everything is an expression, how it is stateless, etc. But we'll get to all of that eventually. If you get this, everything else will follow naturally (mostly).

## 2.3 Combinatory logic

Combinatory logic is a notation to eliminate the need for quantified variables in mathematical logic [13]. That basically means a form of logic without values, just like with lambda calculus, but just pure logical expressions, using so called *combinators*. The idea of combinators first came from Schönfinkel [8], and was later rediscovered by Curry [3]. Combinators are just symbols, in this case letters, that perform operations on symbols that succeed it. We've actually looked at one of these combinators already.

We will be using Curry's names for combinators, since his names are most widely used.

### 2.3.1 Identity

The first combinator we'll cover is $I$. It does exactly the same thing as our $I$ function we defined in lambda calculus in the previous subsection ($I := \lambda a.a$). In fact, all combinators can be defined in lambda calculus. Lambda calculus is really just 90% combinatory logic, but without identifiers. This combinator may seem quite useless, but it is actually quite useful when composing combinators, which we'll come to soon.

### 2.3.2 The omega combinator

The next combinator we'll cover is $M$. All it does is repeat its one argument twice. It can be defined in lambda calculus as follows:

$$M := \lambda f.ff$$

We could, for example, look at what happens when you apply $M$ to $I$. We get:

$$MI = II = I$$

[5] That's what makes them higher-order (and first-class).

Or, written out in lambda calculus:

$$(\lambda f.ff)\lambda a.a = (\lambda a.a)\lambda a.a = \lambda a.a$$

What happens if you apply $M$ to $M$? You get:

$$MM = MM = MM = ...$$

ad infinitum. Or in lambda calculus:

$$(\lambda f.ff)\lambda f.ff = (\lambda f.ff)\lambda f.ff = (\lambda f.ff)\lambda f.ff = ...$$

This expression cannot be evaluated. We say that it doesn't have a $\beta$-normal form. In lambda calculus and combinatory logic not every expression is reducable. As we've seen in the second to last paragraph of section 2.1: there is no single algorithm to decide wether a lambda expression has a $\beta$-normal form[6].

$MM$ is sometimes called the $\Omega$ combinator. Omega, because it is the end of the Greek alphabet. The $M$ combinator is sometimes called the $\omega$ combinator because of this. Combinators often have many different names. Sometimes because scientists discovered them seperatly, unaware of eachother, sometimes because they preferred a different name, sometimes because scientists like to give them pet names [7].

### 2.3.3 The constant combinator

The next combinator we'll cover is $K$. It is a combinator that takes two arguments and returns the first. We can easily define it in lambda calculus as follows:

$$K := \lambda ab.a$$

Remember that we defined this as a Curry'd function. This means we can give it just one argument and get a new function out of it. Let's say we apply $K$ to 5:

$$K5 = (\lambda ab.a)5 = \lambda b.5$$

Our new function, $K5$, is a function that takes an argument and returns 5. This means that whatever we apply this function to, we always get 5. $K$ gets its name from the German word *Konstant*, meaning constant. You can probably see why.

Just like with the previous combinators, it'll prove very useful, much more so than you'd expect.

### 2.3.4 The kite

Here is where things get a little spicier. Our next combinator is $KI$. It takes two arguments and returns the latter. We can define it in lambda calculus as follows:

$$KI := \lambda ab.b$$

You may already be thinking about its name. Why does it have two letters? And why are they two letters we've talked about already? Well, the answer is very simple. If you apply $K$ to $I$, you get $KI$. Don't believe me? Let's try!

If we use our definition of $KI$ and apply it to $xy$ we get:

$$KIxy = (\lambda ab.b)xy = y$$

[6] This is not exactly what is written, but it means the same thing.

[7] I have a theory they are just trying to throw us off

But if we use the K and I combinators seperately, we get the following:

$$KIxy = (\lambda ab.a)Ixy = (\lambda b.I)xy = Iy = y$$

If you think about it, it is very logical. If $K$ takes two arguments and returns the first, then, in this case, it uses up both $I$ and $x$ and returns $I$, which will just return the next argument, in this case $y$. $KI$ will always return the second symbol after the I, because—again—the first gets used up by $K$.

We can also just see what function we get when we apply $K$ to $I$:

$$KI = (\lambda ab.a)I = \lambda b.I = \lambda b.\lambda a.a = \lambda ba.a$$

We get our definition of $KI$ (except the names of the arguments are switched).

We're starting to define combinators as combinations of other combinators. Every combinator, in fact, can be defined as a combination of other combinators. That's why they are called combinators.

### 2.3.5 The flip combinator

The next combinator is is the $C$ combinator. The $C$ combinator is definable in lambda calculus as:

$$C := \lambda fab.fba$$

What it basically does is switch the arguments to the next combinator around.

If we apply $C$ to $K$ and two random symbols, we get the same result we would get if we had applied $KI$ to those same symbols:

$$CKxy = Kyx = y$$

$$KIxy = y$$

$$CK = KI$$

Let's see what happens when we apply C to K in lambda calculus (I have changed the names of $K$'s arguments as to avoid confusion with those of $C$):

$$(\lambda fab.fba)\lambda xy.x = \lambda ab.(\lambda xy.x)ba$$

We don't get our exact definition of $KI$. But we can see that for every input, $CK$ and $KI$ *always* produce the same output. We say that these functions are *extensionally equal*—they have been defined seperately and we cannot rewrite one to the other, but we know that they produce the same results, so they must be equal.

You can do the same thing to find out that $CKI = K$. It really does make sense. $K$ and $KI$ both "select" one of two arguments. One selects the first, the other selects the latter. Flipping their arguments make them select the opposite of what they normally would, so they select the argument that the other combinator usually would.

### 2.3.6 The composition combinator

Our next combinator, $B$, is defined as follows:

$$B := \lambda fga.f(ga)$$

It applies a to b before applying f to the result. This combinator is used for function composition. This function applies $g$ on $a$ and applies $f$ on that. We say that this function composes $f$ with $g$.

In mathematics we usually write function composition as follows:

$$f \circ g$$

So

$$(f \circ g)a = f(ga)$$

In combinatory logic we write function composition as follows:

$$Bfg$$

which reduces to

$$(\lambda fga.f(ga))fg = \lambda a.f(ga)$$

in lambda calculus

There is something noteworthy about function composition: function composition works from right to left. When you compose functions, the rightmost function get's called first, instead of the leftmost function, which you can easily tell just by looking at the definition of $B$.

### 2.3.7 The thrush

Our next combinator is $T_h$. It is defined as follows:

$$T_h := \lambda af.fa$$

It swaps around two functions.

### 2.3.8 The starling

Our last combinator is $S$. It can be defined as follows:

$$S := \lambda fga.fa(ga)$$

It applies f to a and its result to the result of the application of g to a.

### 2.3.9 $S$ and $K$

As I've said, every combinator can be defined as a combination of other combinators. The question arises: how many combinators do we need to define every other combinator? It turns out you need just two. You can define every other combinator using just $S$ and $K$.

### 2.3.10 To Mock a Mockingbird

At the start of this section about combinatory logic, I said that Schönfinkel [8] invented combinatory logic as a way of removing the need for quantified variables. He started with propositional logic and stripped it down until there was a very pure and simple form of logic left. But how can we use this form of logic in the real world, if he even removes things like propositions? You already know that it is Turing complete, so it must be able to do any computation, but I haven't explain how yet (see section 3). But we can use combinatory logic in the real world already.

You may have noticed some of the previous subsections have birdnames as titles. This is because they are the names given to the combinators, discussed in the respective subsections, by an author named Smullyan. He is a mathematician who likes to write puzzle books. His book *To Mock a Mockingbird* [9] is practically a large metaphor for combinatory logic. There are some unrelated puzzles in the beginning of the book, but the rest is about a big forest with birds. The birds represent the combinators. The beginletters of the birdnames are the names of the combinators. The way the birds interact reflects the way the combinators interact. The

reason he chose birds for his metaphor is that Curry was an avid bird watcher. If you feel like you still don't understand the notation of combinatory logic completely, I would recommend you give this book a read, because it explains it very simply and clearly.

In Smullyan's world, there is a forest with birds. These birds represent combinators. If you call out a birdname to a bird, it will give you the name of another bird. If you give a bird a birdname of a bird that is present, it will give you back a name of a bird that is present. Calling out birdnames to birds represents application.

I think it would be fun if we had a look at one of the puzzles to see if we can solve it using our newfound knowledge of combinatory logic. I think the first puzzle is sufficiently interesting. So far, Smullyan has only introduced the mockingbird, which is the omega operator (section 2.3.2), and the idea of function composition, but not yet the bluebird, which is the composition operator (section 2.3.6). I have taken the puzzle directly from the book:

---

It could happen that if you call out $B$ to $A$, $A$ might call the same bird $B$ back to you. If this happens, it indicates that $A$ is fond of the bird $B$. In symbols, $A$ is fond of $B$ means that: $AB = B$

We are now given that the forest satisfies the following two conditions.

$C_1$ (the composition condition): For any two birds $A$ and $B$ (whether the same or different) there is a bird $C$ such that for any bird $x$, $Cx = A(Bx)$. In other words, for any birds $A$ and $B$ there is a bird $C$ that composes $A$ with $B$.

$C_2$ (the mockingbird condition): The forest contains a mockingbird $M$.

One rumor has it that every bird of the forest is fond of at least one bird. Another rumor has it that there is at least one bird that is not fond of any bird. The interesting thing is that it is possible to settle the matter completely by virtue of the given conditions $C_1$ and $C_2$.

Which of the two rumors is correct?

---

Do note that in this case, $C$ and $B$ do not refer to the $C$ and $B$ combinators we've looked at previously.

The answer will be shown on the next page.

Because of $C_1$ and $C_2$, we know that for every bird $A$, there's a bird—we'll call it $C$—that composes $A$ with $M$. We can say the following:

$$Cx = A(Mx) = A(xx)$$

If we now fill in $C$ in place of $x$, we get:

$$A(CC) = CC$$

We thus know that for any bird $A$, $A$ is fond of the bird $CC$, where $C$ composes $A$ with $M$. Therefore rumour one is true and rumour two is false.

The answer Smullyan [9] gives is a bit more verbose, but it comes down to the same thing.

# 3 Using lambda calculus for computation

Now that we've gone throught lambda calculus and combinatory logic, it is finally time to get to the good parts. I hope the way here wasn't too boring or difficult. We will now look at how to use lambda calculus for computation.

## 3.1 Church encoding

To do computation with lambda calculus, we need to be able to do a few things such as boolean logic and arithmetic. Lambda calculus itself doesn't have these things built in to it, but we can define things such as boolean logic and arithmetic in lambda calculus. In section 2.1, I talked about how mathematicians were defining everything in maths. We can build of of their work and see how we can implement their definitions in lambda calculus. This is exactly what Church did.

### 3.1.1 Church booleans

Let's look at a simple form of computation before jumping into arithmetic. We'll start with binary logic. In it's simplest form, binary logic is really just control flow: *if A then B else C* ($A\,?\,B\,:\,C$). We want to have some condition that chooses between two expressions. We know how to do that already (see section 2.3). $K$ choses the first of two expressions and $KI$ the latter. We can define true to be $K$ and false to be $KI$:

$$T := K = \lambda ab.a$$

$$F := KI = \lambda ab.b$$

But how can we do logic gates? Let's look at negation. We have already looked at the flip combinator—$C$ (section 2.3.5), which does exactly what we want. Thus, we can say:

$$NOT := C = \lambda fab.fba$$

We could also do it another way. We already know how to do control flow, so we could define a function that chooses the opposite of what the input is. In simple programming terms:

$$if\ p\ then\ F\ else\ T$$

Or in a C-like expression:

$$p\ ?\ F\ :\ T$$

Or in lambda calculus:

$$NOT := \lambda p.pFT$$

It depends which one's preferable. The $C$ combinator is a bit more elegant and performat, because it takes less function applications, but you get a function that is only extensionally equal to $T$ or $F$, while the other definition literally returns $T$ or $F$.

How do we define $AND$? We could define it in simple programming terms, and then translate it to lambda caluclus. In simple programming terms, $AND$ would look like this:

$$if\ p\ then\ (if\ q\ then\ T\ else\ F)\ else\ f$$

$$p\ ?\ q\ ?\ T : F : F$$

In lambda calculus we would get:

$$AND := \lambda pq.p(qTF)F$$

But this is a very naïve way of defining $AND$. If you look closely at the $qTF$ part, you notice that it actually just returns whatever $q$ is anyway. Thus, we can say:

$$AND := \lambda pq.pqF$$

You can also replace the remaining $F$ with $q$ if you want, because the $F$ will only be returned if $q$ is $F$—in other words: $q$ in that case is alway false. So we can say:

$$AND := \lambda pq.pqp$$

This is the equivalent of:

$$if\ p\ then\ q\ else\ p$$

$$p\ ?\ q\ :\ p$$

We can define $OR$ in a very similar way. If the first argument is true, we just return it, else we return the second argument:

$$OR := \lambda pq.ppq$$

Defining $XOR$ is very easy too. If the first argument is true, then we want to return what the second argument is not, else we want to return what the second argument is. In lambda calculus:

$$XOR := \lambda pq.p(NOT\ q)q$$

Giving:

$$XOR := \lambda pq.p(qFT)q$$

Alternatively:

$$XOR := \lambda pq.p((\lambda fab.fba)q)q = \lambda pq.p(\lambda ab.qba)q$$

We can define boolean equality ($BEQ$) similarly:

$$BEQ := \lambda pq.pq(qFT)$$

Or:

$$BEQ := \lambda pq.pq(\lambda ab.qba)$$

Which you can read as: "If $p$ is true, then return what $q$ is, else return what $q$ is not."

Let's look at an example expression and solve it in lambda calculus. We'll solve the following expression:

$$!\,(x\ \&\&\ y)\ == !\,x\ ||\ !\,y$$

If you're not familiar with C-like expressions: ! means NOT, && means AND, == means BEQ and || means OR. ! has a higher and == a lower precedence than the rest.

We can write and reduce this expression using the functions/combinators we defined (using $NOT := \lambda p.pFT$, because else we wouldn't be able to write out $NOT$ without using lambda calculus):

$$\begin{aligned}
& BEQ\ \ (NOT\ (AND\ x\ y))\ \ (OR\ (NOT\ x)\ (NOT\ y)) \\
=\ & BEQ\ \ (NOT\ (xyx))\ \ (OR\ (xFT)\ (yFT)) \\
=\ & BEQ\ \ (xqxFT)\ \ (xFT\ (xFT)\ (yFT)) \\
=\ & xyxFT\ \ (xFT\ (xFT)\ (yFT))\ \ (xFT\ (xFT)\ (yFT)\ FT)
\end{aligned}$$

If we've done this correctly, then according to *De Morgan's Law*, we should always get $T$ for any substitution of $x$ and $y$ for $T$ and $F$ in this final expression.

$x = F$ and $y = F$ gives us:

$$\begin{aligned}
& FFFFT\ \ (FFT\ (FFT)\ (FFT))\ \ (FFT\ (FFT)\ (FFT)\ FT) \\
=\ & FFT\ \ (TTT)\ \ (TTTFT) \\
=\ & TT\ \ (TFT) \\
=\ & TTF \\
=\ & T
\end{aligned}$$

$x = F$ and $y = T$ gives us:

$$\begin{aligned}
& FTFFT\ \ (FFT\ (FFT)\ (TFT))\ \ (FFT\ (FFT)\ (TFT)\ FT) \\
=\ & FFT\ \ (TTF)\ \ (TTFFT) \\
=\ & TT\ \ (TFT) \\
=\ & TTF \\
=\ & T
\end{aligned}$$

$x = T$ and $y = F$ gives us:

$$\begin{aligned}
& TFTFT\ \ (TFT\ (TFT)\ (FFT))\ \ (TFT\ (TFT)\ (FFT)\ FT) \\
=\ & FFT\ \ (FFT)\ \ (FFTFT) \\
=\ & TT\ \ (TFT) \\
=\ & TTF \\
=\ & T
\end{aligned}$$

$x = T$ and $y = T$ gives us:

$$\begin{aligned}
& TTTFT\ \ (TFT\ (TFT)\ (TFT))\ \ (TFT\ (TFT)\ (TFT)\ FT) \\
=\ & TFT\ \ (FFF)\ \ (FFFFT) \\
=\ & FF\ \ (FFT) \\
=\ & FFT \\
=\ & T
\end{aligned}$$

We can also try to solve this expression in lambda calculus (using the shorthand notation

and using $NOT := C$, because it is easier to write out):

$$\begin{aligned}
& \lambda xy.(\lambda pq.pq(\lambda cd.qdc))((\lambda fab.fba)((\lambda pq.pqp)xy))((\lambda pq.ppq)((\lambda fab.fba)x)((\lambda fab.fba)y)) && (1)\\
= \ & \lambda xy.(\lambda pq.pq(\lambda cd.qdc))((\lambda fab.fba)(xyx))((\lambda pq.ppq)(\lambda ab.xba)(\lambda ab.yba)) && (2)\\
= \ & \lambda xy.(\lambda pq.pq(\lambda cd.qdc))(\lambda ab.xyxba)((\lambda ab.xba)(\lambda ab.xba)(\lambda ab.yba)) && (3)\\
= \ & \lambda xy.(\lambda pq.pq(\lambda cd.qdc))(\lambda ab.xyxba)(x(\lambda ab.yba)(\lambda ab.xba)) && (4)\\
= \ & \lambda xy.(\lambda ab.xyxba)(x(\lambda ab.yba)(\lambda ab.xba))(\lambda cd.x(\lambda ab.yba)(\lambda ab.xba)dc) && (5)\\
= \ & \lambda xy.xyx(\lambda cd.x(\lambda ab.yba)(\lambda ab.xba)dc)(x(\lambda ab.yba)(\lambda ab.xba)) && (6)\\
\equiv \ & \lambda xy.xyx(\lambda cd.x(\lambda ab.yba)(\lambda ab.a)dc)(x(\lambda ab.yba)(\lambda ab.a)) && (7)\\
\equiv \ & \lambda xy.xyx(xy(\lambda ab.b))(x(\lambda ab.yba)(\lambda ab.a)) && (8)\\
\equiv \ & \lambda xy.xyx(xy(\lambda ab.b))(\lambda ab.a) && (9)\\
\equiv \ & \lambda xy.xyx(\lambda ab.a)(\lambda ab.a) && (10)\\
\equiv \ & \lambda ab.a && (11)
\end{aligned}$$

We know that expressions 6 and 7 are extensionally equal;

$$\lambda x.x(\lambda ab.yba)(\lambda ab.xba) \equiv \lambda x.x(\lambda ab.yba)(\lambda ab.a)$$

because $x$ picks between $\lambda ab.yba$ and $\lambda ab.xba$. $\lambda ab.xba$ only gets picked when $x$ is $F$, so $\lambda ab.xba$ is always $\lambda ab.Fba{=}\lambda ab.a$. Therefore, a $x(\lambda ab.yba)(\lambda ab.xba)$ that occurs at the start of its grouping (we do need to keep left-associativity in mind) can be replaced with $x(\lambda ab.yba)(\lambda ab.a)$.

We also know that expressions 7 and 8 are extensionally equal. We know that

$$\lambda xcd.x(\lambda ab.yba)(\lambda ab.a)dc$$

selects between

$$\lambda cd.(\lambda ab.yba)dc = \lambda cd.ycd \equiv y$$

and

$$\lambda cd.(\lambda ab.a)dc = \lambda cd.d = \lambda ab.b$$

therefore

$$\lambda xcd.x(\lambda ab.yba)(\lambda ab.a)dc \equiv \lambda x.xy(\lambda ab.b)$$

Expressions 8 and 9 are also extensionally equal, because $xyx$ selects between the expressions $xy(\lambda ab.b)$ and $x(\lambda ab.yba)(\lambda ab.a)$. The expression $x(\lambda ab.yba)(\lambda ab.a)$ only gets selected if $xyx = F$. There are two ways for $xyx = F$ to be true; either $x = F$, or both $x = T$ and $y = F$. $x = F$ gives

$$x(\lambda ab.yba)(\lambda ab.a) = F(\lambda ab.yba)(\lambda ab.a) = \lambda ab.a$$

and $x = T$ and $y = F$ gives

$$x(\lambda ab.yba)(\lambda ab.a) = T(\lambda ab.Fab)(\lambda ab.a) = \lambda ab.Fab = \lambda ab.a$$

meaning that for all the posibilities of

$$xyx(xy(\lambda ab.b))(x(\lambda ab.yba)(\lambda ab.a)) = x(\lambda ab.yba)(\lambda ab.a)$$

also

$$x(\lambda ab.yba)(\lambda ab.a) \equiv \lambda ab.a$$

meaning

$$\lambda xy.xyx(xy(\lambda ab.b))(x(\lambda ab.yba)(\lambda ab.a)) \equiv \lambda xy.xyx(xy(\lambda ab.b))(\lambda ab.a)$$

We can apply the same reasoning to prove that equations 9 and 10 are extensionally equal. We know $xyx$ picks between $xy(\lambda ab.a)$ and $\lambda ab.a$. For $xyx$ to pick the first, $xyx$ must equal $T$, which is only possible if both $x = T$ and $y = T$. Which gives

$$xy(\lambda ab.a) = TT(\lambda ab.a) = T = \lambda ab.a$$

The last step should be quite self-explainatory; $xyx$ can only select between $\lambda ab.a$ and $\lambda ab.a$.

### 3.1.2 Church numerals

Now that we've covered boolean logic, it is finally time to move on to arithmetic. Remember that in section 2.1, I said that Peano [6] defined natural numbers? He basically defined zero and then defined every number as a successor of the previous number, starting from zero. He also defined the properties of natural numbers.

We can do exactly the same in lambda calculus. The key concept is function composition (section 2.3.6).

We'll define functions that compose a given function $f$, $n$ times with themselves, where $n$ is the natural number the function is supposed to represent. Thus we'll say

$$\begin{aligned}
N0 &:= \lambda fa.a \\
N1 &:= \lambda fa.fa \\
N2 &:= \lambda fa.f(fa) \\
N3 &:= \lambda fa.f(f(fa)) \\
N4 &:= \lambda fa.f(f(f(fa)))
\end{aligned}$$

and so on, or in a more standard mathematical notation

$$\begin{aligned}
0f &:= 0 \\
1f &:= f \\
2f &:= f \circ f \\
3f &:= f \circ f \circ f \\
4f &:= f \circ f \circ f \circ f
\end{aligned}$$

You can also say

$$\begin{aligned}
N0 &:= F \\
N1 &:= \lambda f.f = I \\
N2 &:= \lambda f.Bff \\
N3 &:= \lambda f.B(Bff)f \\
N4 &:= \lambda f.B(B(Bff)f)f
\end{aligned}$$

We can't define an infinite number of functions by hand, so we'll use this shorthand definition:

$$n := \lambda fa.f^{\circ n}a$$

meaning: applying a number $n$ to a function $f$ is the same as composing $f$ with $f$, $n$ times. This is meaningful, because it allows us to do something a given number of times.

We can use this in an example. Let's say we wanted to do a negation multiple times. We can use our newly defined numbers:

$$\begin{aligned}
&N0\ \ NOT\ \ T = T \\
&N1\ \ NOT\ \ T = NOT\ \ T = F \\
&N2\ \ NOT\ \ T = NOT\ \ (NOT\ \ T) = T \\
&N3\ \ NOT\ \ T = NOT\ \ (NOT\ \ (NOT\ \ T)) = F
\end{aligned}$$

Numbers are just function composition. Function composition, and thus the $B$ combinator (section 2.3.6), is fundamental to all arithmetic in lambda calculus. What happens when we

compose two numbers, say $N3$ with $N3$?

$$\begin{aligned}
& B\ \ N3\ \ N3 \\
=\ & (\lambda fgh.f(gh))\ N3\ \ N3 \\
=\ & \lambda h.N3\ (N3\ \ h) \\
=\ & \lambda h.N3\ ((\lambda fb.f(f(fb))h)) \\
=\ & \lambda h.N3\ (\lambda b.h(h(hb))) \\
=\ & \lambda h.(\lambda fa.f(f(fa)))(\lambda b.h(h(hb))) \\
=\ & \lambda h.\lambda a.(\lambda b.h(h(hb)))((\lambda b.h(h(hb)))((\lambda b.h(h(hb)))a)) \\
=\ & \lambda ha.(\lambda b.h(h(hb)))((\lambda b.h(h(hb)))(h(h(ha)))) \\
=\ & \lambda ha.(\lambda b.h(h(hb)))(h(h(h(h(h(ha)))))) \\
=\ & \lambda ha.h(h(h(h(h(h(h(h(ha))))))))) \\
=\ & N9
\end{aligned}$$

We can tell that composition with natural numbers is the same as multiplication, which makes a lot of sense, because function composition is associative. The ninefold composition of $f$ is the same as the threefold composition of the threefold composition of $f$:

$$f \circ f \circ f \circ f \circ f \circ f \circ f \circ f \circ f = (f \circ f \circ f) \circ (f \circ f \circ f) \circ (f \circ f \circ f)$$

We know that if we were to define a $MULT$ combinator, the following should be true:

$$MULT\ \ N3\ \ N3\ \ f\ \ a = (f \circ f \circ f \circ f \circ f \circ f \circ f \circ f \circ f)a$$

and in extension

$$\begin{aligned}
MULT\ \ N3\ \ N3\ \ f\ \ a &= ((f \circ f \circ f) \circ (f \circ f \circ f) \circ (f \circ f \circ f))\ a \\
&= ((N3\ \ f) \circ (N3\ \ f) \circ (N3\ \ f))\ a \\
&= N3\ (N3\ \ f)\ a \\
MULT\ \ N3\ \ N3\ \ f\ \ &= N3\ (N3\ \ f) \\
&= B\ \ N3\ \ N3\ \ f \\
MULT &= B
\end{aligned}$$

thus we can say

$$MULT := B = \lambda nkf.n(kf)$$

What happens when we apply $N3$ to $N2$? We get the following:

$$\begin{aligned}
& N3\ \ N2 \\
=\ & \lambda fa.f(f(fa))\ N2 \\
=\ & \lambda a.N2(N2(N2\ \ a)) \\
=\ & \lambda a.N2(N4\ \ a)) \\
=\ & \lambda a.N8\ \ a \\
\equiv\ & N8
\end{aligned}$$

in other "words"

$$\begin{aligned}
& 3\ \ 2 \\
=\ & 2 \circ 2 \circ 2 \\
=\ & 8
\end{aligned}$$

so application of natural numbers is the same exponentiation, but the numbers are reversed. Thus, we can say:

$$POW := \lambda nk.kn$$

which is just our $T_h$ combinator (section 2.3.7).

$$POW := T_h$$

How do we do addition? Let's start with adding one—in other words—finding the successor. All we need to do is compose the function once more. We could define the $SUCC$ function as follows:

$$SUCC := \lambda nfa.f(nfa)$$

Alternatively:

$$SUCC := \lambda nf.Bf(nf)$$

which some find prettier, because it makes it clear that we're doing function composition, but it also takes a bit longer to compute, because the second definition reduces down to the first:

$$\begin{aligned} & \lambda nf.Bf(nf) \\ = \ & \lambda nf.(\lambda gha.g(ha))f(nf) \\ = \ & \lambda nf.\lambda a.f((nf)a) \\ = \ & \lambda nfa.f(nfa) \end{aligned}$$

so if you use the second, you need to perform that reduction every time you invoke it.

If we now try to find the successor of $N3$, we get

$$\begin{aligned} & SUCC \ \ N3 \\ = \ & (\lambda nfa.f(nfa)) \ N3 \\ = \ & \lambda fa.f(N3 \ \ fa) \\ = \ & \lambda fa.f((\lambda hb.h(h(hb)))fa) \\ = \ & \lambda fa.f(f(f(fa))) \\ = \ & N4 \end{aligned}$$

or

$$\begin{aligned} & SUCC \ \ N3 \\ = \ & (\lambda nf.Bf(nf)) \ N3 \\ = \ & \lambda f.Bf(N3 \ \ f) \\ = \ & \lambda f.Bf((\lambda ha.h(h(ha)))f) \\ = \ & \lambda f.Bf(\lambda a.f(f(fa))) \\ = \ & \lambda f.Bf(\lambda a.f(f(fa))) \\ = \ & \lambda f.(\lambda ghb.g(hb))f(\lambda a.f(f(fa))) \\ = \ & \lambda f.(\lambda b.f((\lambda a.f(f(fa)))b)) \\ = \ & \lambda f.\lambda b.f(f(f(fb))) \\ = \ & \lambda fa.f(f(f(fa))) \\ = \ & N4 \end{aligned}$$

To add two numbers, we can now just call this successor function multiple times. Say we wanted to add three to four. We can call the successor function three times on four:

$$\begin{aligned} & N3\ \ SUCC\ \ N4 \\ = \ & SUCC\,(SUCC\,(SUCC\ \ N4)) \\ = \ & SUCC\,(SUCC\ \ N5) \\ = \ & SUCC\ \ N6 \\ = \ & N7 \end{aligned}$$

Thus we can say

$$ADD := \lambda nk.(n\ \ SUCC\ \ k)$$

$SUCC$ is really just an infix $ADD$.

We could also say that if we want to add two numbers $n$ and $k$, we can compose a function $n$ times and $k$ times and compose the results to get a $(n+k)$-fold composition of the function. In lambda calculus:

$$ADD := \lambda nkf.B(nf)(kf)$$

or

$$ADD := \lambda nkfa.(nf)(kfa)$$

which looks a lot like our $SUCC$ function, just with an extra argument to decide the number of times to compose the function, and it's prefix instead of infix. In a more mathematical notation, you can say

$$ADD\ \ n\ \ k\ \ f := f^{\circ n} \circ f^{\circ k}$$

### 3.1.3 Merging of boolean operations and arithmetic

The C programming language has only added booleans in the C99 standard. In C, booleans are really just integers. All boolean expressions are really just arithmetic. In C, $false$ is synonymous with 0 and every nonzero number means $true$. In lambda calculus, this is the same. Our definition of $N0$ is exactly the same as our definition of $F$. In this section, we're going to talk about functions that are both useful in arithmetic and boolean logic, which are really just the same.

I want to define a function that takes a church numeral and returns $T$ if it's $N0$ or $F$ if it's not. I will call this function $ISZERO$. To do this, we need to remember what a church numeral really is; it's a function that takes a function and an argument and applies that function a given number of times to that argument. The unique thing about $N0$ is that it applies that given function zero times to the given argument, meaning it just returns the argument. That means that if we give $T$ as the second argument, $N0$ will return $T$. So far, we have this:

$$ISZERO := \lambda n.n...T$$

We still need to fill in the dots. If $n$ isn't $N0$, whatever is on the dots will be applied to $T$ $n$ times. We want this to always return $F$. We know a function that can do this: the constant function (section 2.3.3). Thus, we get:

$$ISZERO := \lambda n.n(KF)T$$

Here is an example for if it is not yet clear to you why we use the constant combinator:

$$ISZERO\ \ N3 = (\lambda n.n(KF)T)N3 = N3(KF)T = K\ \ F\ \ (KF(KF\ \ T)) = F$$

We've covered addition, multiplication and exponentiation, which were quite simple, but what about subtraction, division and finding the square root? Well, in contrast to the operations

we've covered, these are actually quite complex. We'll quickly look at subtraction, but it gets very complicated very quickly, and it's really out of the scope of this text to cover all of arithmetic in lambda calculus. I just wanted to show that it's possible and that lambda calculus is turing complete.

Just like we defined a successor function before defining addition, we'll define a predecessor function first. The predecessor function can be defined as:

$$PRED := \lambda n.n\ (\lambda g.ISZERO\ (g\ \ N1)\ I\ (B\ \ SUCC\ \ g))\ (K\ \ N0)\ N0$$

This is quite a bit to digest.

#### 3.1.4 Data structures

Pairs and linked lists.

### 3.2 Recursion

Loops, y-combinator.

## 4 Functional programming (lambda calculus applied)

### 4.1 Why do we even care?

### 4.2 Lambda functions

### 4.3 Laziness

### 4.4 Types

### 4.5 Monads

### 4.6 Haskell

### 4.7 Comparison to with other paradigms

#### 4.7.1 Declarative vs imperative

#### 4.7.2 Usefulness vs conceptual purity

#### 4.7.3 Meta programming

## 5 Functional programming in other paradigms

### 5.1 Lambda functions

### 5.2 Iterators

## 6 Possibilities for the future

### 6.1 Conceptual *and* useful

## Afterword

## References


[1] Alonzo Church. "A set of postulates for the foundation of logic". In: *Annals of mathematics* (1932), pp. 346–366.

[2] Alonzo Church. "An unsolvable problem of elementary number theory". In: *American journal of mathematics* 58.2 (1936), pp. 345–363.

[3] Haskell Brooks Curry. "Grundlagen der kombinatorischen Logik". In: *American journal of mathematics* 52.4 (1930), pp. 789–834.

[4] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Nebert, 1879.

[5] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für mathematik und physik* 38.1 (1931), pp. 173–198.

[6] Giuseppe Peano. *Arithmetices principia: Nova methodo exposita*. Fratres Bocca, 1889.

[7] Bertrand Russell and Alfred North Whitehead. *Principia mathematica to* 56*. Vol. 2. Cambridge University Press Cambridge, UK, 1997.

[8] Moses Schönfinkel. "Über die Bausteine der mathematischen Logik". In: *Mathematische annalen* 92.3-4 (1924), pp. 305–316.

[9] Raymond Merrill Smullyan. *To Mock a Mockingbird: and other logic puzzles including an amazing adventure in combinatory logic*. Oxford University Press, USA, 2000.

[10] Alan Mathison Turing. "Computability and $\lambda$-definability". In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163.

[11] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5.

[12] Alan Mathison Turing. *On computable numbers, with an application to the Entscheidungsproblem; a correction*. Royal Society, 1937.

[13] Wikipedia. *Combinatory logic — Wikipedia, The Free Encyclopedia*. 2020. URL: `https://web.archive.org/web/20200920164914if_/https://en.wikipedia.org/wiki/Combinatory_logic` (visited on 20/09/2020).

[14] Wikipedia. *Gödel's incompleteness theorems — Wikipedia, The Free Encyclopedia*. 2020. URL: `https://web.archive.org/web/20200927072027/https://en.wikipedia.org/wiki/G%C3%B6del%27s_incompleteness_theorems` (visited on 27/09/2020).