# The Lambda Calculus Without Tears (Sample)

## Contents



# Background

The Lambda Calculus was a topic I struggled with during my undergraduate degree. Whilst the effort I invested in the topic paid off in a big way, I felt hampered by a lack of basic working examples for each new topic I was introduced to.

There were only two books available on the subject, both of which described the topic mathematically without any basic examples to ease someone into it. Given the importance of the Lambda Calculus within

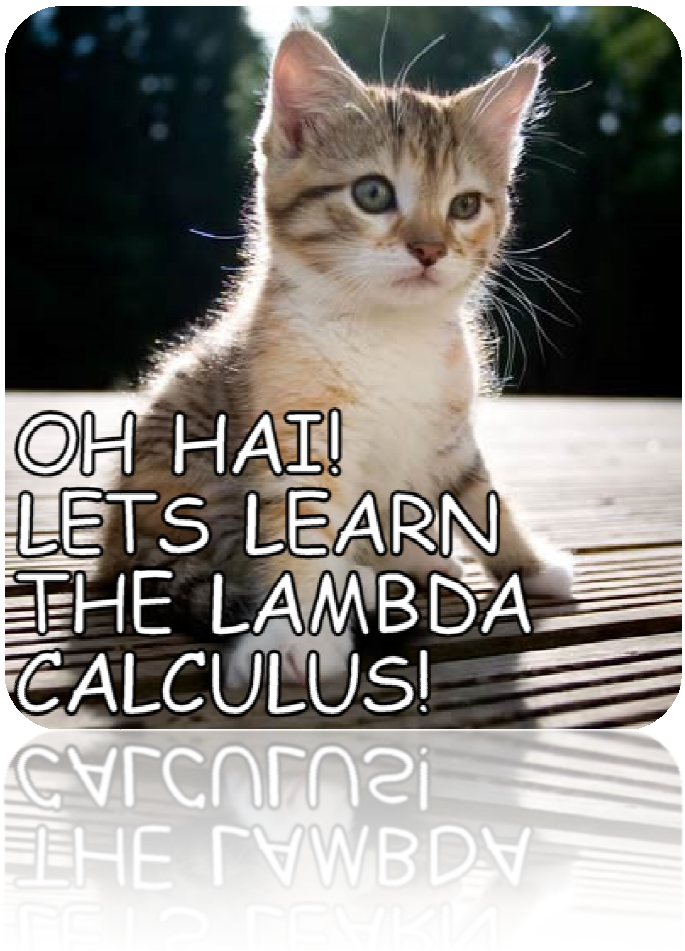
theoretical computer science, this felt wrong to me.

After the topic clicked in my head, I passed the exams and wanted to write my own guide to the subject for people like me who work best through examples and real-word explanations.

I began on the first chapter of this endeavour just after I graduated, but never found time to return to it. Then, there was a burglary in my London flat where I lost the original Pages document. Thankfully, I'd emailed a PDF copy of the work to date to a couple of friends which is now all that remains of this project.

So, in lieu of me ever continuing with it I present the first and only completed chapter, and a little bit of the second chapter. Happy reading!

## Part 1: Getting the Vibe




In this first chapter, we're going to begin by just getting a feel for what the lambda calculus is. By the end of this chapter, you should have a grasp of what a lambda calculus expression looks like, what it does and how it works. With any luck, you will manage this without becoming an alcoholic.

### Start Simple

Let's kick off with the simplest of terms in the lambda calculus:

$$\lambda x.x$$

It takes in "something" and gives the same "something" back. So, the stuff between the Greek letter "$\lambda$" (lambda) and the "." represents the input, and the stuff after the "." represents our output. So, if I put in a **cookie**, I get back a **cookie**. Simple enough, so what about this:

$$\lambda xy.x$$

Can you hazard a guess at what it does? Well, it takes in two inputs (which it names $x$ and $y$), and it gives you back the $x$. The $y$ is never seen again. The In other words, if I put in a **cookie** and a **doughnut** (in that order), I only get the **cookie** back (What a waste of a doughnut). How about this one:

$$\lambda xy.yx$$

This takes in two values ($x$ and $y$), and returns them both, but in reverse order. Therefore, if we put in a **cookie** and a **doughnut**, we'd get back a **doughnut** and a **cookie**.

But, wouldn't it be better to have a term which let me put in a **cookie**, and gave me two back? Well, we can! And, here's how:

$$\lambda x.xx$$

This term takes in one $x$, and gives you back two. So, if we input **one cookie**, we get **two** back! OK, with that in mind, there's one more case we need to look at:

$$\lambda x.xz$$

This term accepts on input (an $x$) and returns that $x$ and also a $z$. So, if we put in a **cookie**, we get back a **cookie** and a $z$. I've no idea what a $z$ will taste like though!

## Semantic Hell

Next up, it's important to understand that there are varied ways of writing terms. The following two terms are identical; they're just two different ways of writing the same thing:

$$\lambda xy.xy \equiv \lambda x.\lambda y.xy$$

## Bracket Hell

There are invisible brackets in every lambda calculus term. Let's have a look at these two identical lambda calculus terms:

$$xyz \equiv (((x)y)z)$$

Notice how the brackets associate with the leftmost elements in the term? Mathematicians would say that **bracketing is left-associative**. In the world of the lambda calculus, it's **cooler** to be on the **left** than the right. Always remember this, and you'll be a hit at parties! (OK, I might be lying a bit there). Anyway, Let's see another example:

$$xyz \equiv (xy)z$$
$$xyz \not\equiv x(yz)$$

The above example shows another case of bracketing issues. The top two terms **are** identical to each other, but the bottom two terms **are not** identical to each other. Really, burn this idea into your mind, 'cause it's really, really important.

## So What's the Point?

If you've made it this far, well done; let's take a breather and ask "what the hell does all this mean?". Well, all we're doing is creating a mathematical way of representing computer programs. For example, consider the following bit of Java code, which is a method called *giveZero*:

```
1       public int giveZero(int input)
2       {
3               return 0;
4       }
```

We can see that *giveZero* takes in **any number** as input (line 1), but always **returns 0** (line 3), completely ignoring the input. Here's an equivalent function in the lambda calculus:

$$\lambda x.ZERO$$

There we go, it takes in something (which it calls $x$) and returns $ZERO$. It doesn't matter what $x$ is. We could input a **nuclear bomb**, and we'd still get a $ZERO$ back. (don't worry yet about how $ZERO$ is represented in the lambda calculus. Just treat it as another element just like an $x$ or a $y$ for now).

So, have I convinced you that the lambda calculus is simply a mathematical representation for programs? If so, take a breather to digest the following ideas before progressing any further:

- Lambda calculus terms are computer programs (or functions/methods if you prefer).
- Lambda calculus terms can have an input and an output.

Once you feel comfortable with these basic ideas, we're ready to learn how we can run these programs.

# Part 2: Reduction & Conversion

## Beta-Reduction 101

If each lambda calculus term is a program, then how do we run these programs? This is the process known as **reduction**, (or **beta-reduction** to give it its proper title).

Let's have a look at an example. Remember our program from earlier; the one which can duplicate things such as cookies and doughnuts? Here it is again:

$$\lambda x.xx$$

Now, let's say we want to run our program, providing a **doughnut** as an input, here's what the process looks like:

$$(\lambda x.xx)\ doughnut \rightarrow_\beta\ doughnut\ doughnut$$

OK, please **don't panic** like I did when I first saw something like this! I'm going to explain *everything* in the above statement, piece by piece.

To start with, just get the following idea: That the bit to the **left** of the "$\rightarrow_\beta$" is the program before it's run, and the bit to the **right** of the "$\rightarrow_\beta$" is the result of the program after it's finished working its magic.

We already know that the term $\lambda x.xx$ takes in some input, and returns two copies. See how when we apply our term $(\lambda x.xx)$ to a $doughnut$, and then **beta-reduce** the whole thing, it gives us two $doughnut$s back in return? In other words, the **thing** (in this case: $doughnut$) which comes after the bit which is the program (i.e. $(\lambda x.xx)$) is **the value that is being inputted into the program**.

So, what does the "$\rightarrow_\beta$" mean? Well, this can be read as "reduces to", and implies the point where we evaluate the expression on the left to get the one on the right. Or, to put it another way, this is where we **run our program**! Shall we have a look at another example then?

$$(\lambda xyu.xzu)\ pigeon\ hamster\ aligator \rightarrow_\beta\ pigeon\ z\ aligator$$

So, what's going on this time? Our 'program' ($(\lambda xyu.yzx)$) takes in three inputs, and returns three outputs. The inputs are named $x$, $y$ and $u$ respectively. The program returns $x$, $z$ and $u$ in that order.

In the above example, we input $pigeon$, $hamster$ and $aligator$, we get back $pigeon$, $z$ and $aligator$. Can you see what's happening? Our function replaces the middle of the three inputs with $z$. Seeing as $z$ is not defined as an input, it just remains as a $z$.

OK, one more example then:

$$(\lambda xyz.\, yyxx)\ mario\ luigi\ peach\ bowser \rightarrow_\beta\ luigi\ luigi\ mario\ mario\ bowser$$

A little bizarre on first glance? Consider that the function $(\lambda xyz.\, yyxx)$ only accepts **three** arguments, which it names $x$, $y$ and $z$. Now look at the next part: '$mario\ luigi\ peach\ bowser$'. Notice how we try to input **four** arguments into our function? So, when we beta-reduce, our function just ignores the fourth argument ($bowser$), leaving it just where it is, **untouched**.

With all this in mind, can you work out what the following lambda calculus term reduces to? The answer is at the end of the document[i]:

$$(\lambda xyz.\, xuzzyxx)s\ e\ c\ !\rightarrow_\beta\ ?$$

**Small Hint**: Try using a pen and paper to write down the substitutions for the three inputs $x$, $y$ and $z$.
**Bigger Hint**: The $u$ stays the same in the output, because it is not specified as an input parameter.
**Even Bigger Hint:** The '!' will just be added to the end of the output, as the function only accepts three inputs, and we are giving four arguments (the '!' being the fourth).
**Good Advice:** Don't panic, It's way easier than it looks on first glance!

Now, here's the pinch. Because these examples are relatively simple, we can work out how to reduce them in our heads, and intuit what the output should be. Sadly, not all terms are so straight forward, and therefore we need a formal way of performing the beta-reduction, so we can do more complex reductions.

# Solutions to Exercises

[i] $(\lambda xyz.xuzzyxx)s\ e\ c\ !\rightarrow_{\beta}\ s\ u\ c\ c\ e\ s\ s\ !$
The $x$ becomes an 's', the $y$ becomes an 'e', the $z$ becomes a 'c' and the u remains the same. When we substitute all this into $xuzzyxx$ we get a 'success' :-)