A Lambda Calculus Primer

June 12, 2015
Updated: March 31, 2017

> *"You do not really understand something unless you can explain it to your grandmother."*
> *—Albert Einstein*

This post is my attempt to do just that, only that the grandmother here is myself. I firmly believe that unless I try to explain something, will I really understand it. This post takes a very laid-back approach, and avoids very technical topics, unless warranted.

Table of contents

Introduction

What is it?

Lambda calculus is a minimal system for expressing computation that conforms to universal models of computation, hence making it a universal model of computation. In other words, it can be called as one of simplest programming languages, only that it looks and behaves differently from the ones we contemporarily know. Lambda calculus also forms as the basis for the popular functional programming languages in current use now.

Do I need to learn it?

Yes, and no. If you want to understand the underlying mechanisms of how software works, or if you want to build the next great language, or if you just want to appreciate the elegance of its art, then yes. However, if you just want to fly a plane without knowing how it ticks, then no. Seriously though, learn it.

What do we do?

When discussing new concepts, it is very important to layout the axioms or the initial ruleset. Think of it as defining new terms in play, and giving them meaning. The context in which these meanings live are important. For example, for the gardener the hose is used to water the plans, while for the fireman, the hose is used to put out the fire. When the gardener, or the fireman grabs the hose, he will not question what is that he is holding, and what is its purpose. He simply believes in his faith of intuition, to determine the meaning of the hose at the time he grabbed it.

In English, the word “high” has specific meanings. But for all the defined meanings of the word, there is no intrinsic knowledge of the value of the word. We take the meaning as is. We have to agree to use the word in the narrowed context of the users of the word. If we try to deviate from the established meaning of the word, for example, we randomly create a new definition of the word because of whim, chances are it won’t be accepted. We need to believe in the defined connotative and denotative meanings of the word, for it to have meaning to us. The same holds true for lambda calculus—we either accept these axioms and operate in its domain, or we live in Neverland.

Baby steps

Functions

A central player in lambda calculus is the notion of function. Most of us are familiar with functions in our high-level languages, but functions in lambda calculus are slightly different—they need to have at the minimum a single parameter. In most production languages in use now, you can invoke a function that doesn't take an argument. They're usually used for side-effects. In lambda calculus, however, a bare minimum of one argument is enforced. Here's what a minimal function in lambda calculus looks like:

```
λx.x
```

Which is equivalent to:

```
(λz.z)
(λc.c)
```

This equivalence is called the α-conversion. The names do not matter, as long as they're used consistently. Parentheses may be used to remove ambiguity when applying functions. The function above is equivalent to:

```
(λx.x)
```

The Greek letter **λ** denotes that the surrounding context is a function—or something that can be applied or used. The **λ** symbol is used instead of another symbol because of a typesetting issue that is discussed here. So, don't fret too much about, just use it.

What comes next after the **λ** symbol, before the `.`, is the parameter. Technically, it can be any symbol. It simply means the name that can be used when applying that function, to refer to its argument.

The `.` symbol here, is the separator between the parameter list, and the function body. In the function **`(λx.x)`**, the body is simply the symbol **`x`**.

Variables

In lambda calculus, the symbols that are used inside a function are called variables. Going back to the function you defined above,

```
(λx.x)
```

The parameter **`x`** is a variable that is said to be bound, because it sandwiched between **λ** and `.`. However, in the function:

```
(λx.xy)
```

The parameter **y** is a variable that is said to be free, because it does not live between **λ** and ..

Function application

To use a function, you must apply it to something. The bound variables are substituted with what they're applied to—a process called β-reduction.

For example:

```
(λx.x)y
y
```

Let's break it down:

1. Apply **(λx.x)** to **y**:
2. Consume the arguments, then substitute all instances **x** in the body, with **y**.

"Wait, it merely returned the argument y." you may say. That is true. The function **(λx.x)** is the identity function—it is a single-parameter function that returns whatever is was applied to.

Functions are not limited to be applied to symbols. They can also be applied to other functions:

```
(λx.x)(λy.y)
(λy.y)
```

In the example above, the identify function is applied to an identity function, returning an identity function.

Here's another application involving free variables:

```
(λa.ab)(λy.y)
(λy.y)b
b
```

The bound variable **a** was substituted with **(λy.y)**, which is then applied to the free variable **b**, resulting to **b**.

Take note that this function application:

```
(λx.(λy.y))ab
(λy.y)b
b
```

is equivalent to:

```
(λxy.y)ab
b
```

Having multiple parameter names is a shorthand to multiple lambdas, giving the abbreviated version the impression that it consumes multiple arguments at once.

Inside the body of a function, when two symbols are adjacent to one another, the first symbol is presumed to be a function being applied to the second symbol, minus the parentheses. For example, the following code:

```
(λxy.xy)
```

is equivalent to:

```
(λxy.x(y))
```

Let's count!

Start

Since (almost) everything in lambda calculus is expressed as functions, its take on numbers is unique. Arguably, the most important number in lambda calculus is zero (0), which is expressed as:

```
(λsz.z)
```

For convenience purposes, let's label that expression as `0`, with the `≡` symbol read as "is identical to".

```
0 ≡ (λsz.z)
```

Building from `0`, let's enumerate the first three counting numbers:

```
1 ≡ (λsz.s(z))
2 ≡ (λsz.s(s(z)))
3 ≡ (λsz.s(s(s(z))))
```

Successor

The successor of a whole number is defined as the next whole number, counting up, so the successor of `0` is `1`. The definition of the successor function is:

```
S ≡ (λxyz.y(xyz))
```

Let's try that to `0` (in the examples below, the `=` symbol is read as "is reduced to"):

```
S0
≡ (λxyz.y(xyz))(λsz.z)
= (λyz.y((λsz.z)yz))
= (λyz.y((λz.z)z))
= (λyz.y(z))
≡ 1
```

Let's break it down:

1. Determine the successor (S) of zero (0).
2. Spell out the equivalent functional notation.
3. Apply `(λsz.z)` to `y` substituting the bound variable `s` to `y`.
4. Apply `(λz.z)` to `z` substituting the bound variable `z` to `z`.
5. Evaluation stops, and `(λyz.y(z))` is returned, which is the number 1.

Addition

What if you wanted to perform `2+3`? Fortunately, the successor function will do that for you. You express that as `2S3`, where you replace `+` as the infix operator. The addition function is defined as:

```
Name: A
Profile: S ≡ (λxyz.y(xyz))
Inputs: x, y
Outputs: c
Usage: xAy
```

Let's test it out:

```
2+3 ≡ 2A3
≡ (λsz.s(sz))(λxyz.y(xyz))(λuv.u(u(uv)))
= SS3
≡ (λxyz.y(xyz))((λxyz.y(xyz))(λuv.u(u(uv))))
= (λxyz.y(xyz))(λyz.y((λuv.u(u(uv)))yz))
= (λxyz.y(xyz))(λyz.y(y(y(yz))))
≡ S4
= (λyz.y((λyz.y(y(y(yz))))yz))
= (λyz.y(y(y(y(yz)))))
≡ 5
```

Let's break it down:

1. State the problem.
2. Spell out the equivalent functional notations for `2`, `S`, and `3`.
3. Reducing it gives you `SS3`
4. The full version of `SS3`, which corresponds with `2S3` or two `S` and a `3`.
5. Reduce it further.
6. Reduce even further.

7. It is now reduced to **`S4`**.
8. Apply **`S`** to **`4`**.
9. You now arrive at **`5`**.

Multiplication

The multiplication function is defined as:

```
Name: M
Profile: (λxyz.x(yz))
Inputs: a, b
Outputs: c
Usage: Mab
```

Unlike with addition which uses infix syntax, multiplying two numbers follow a prefix syntax. So, to multiply **`2`** and **`3`**, you say **`M23`**.

Let's test that out:

```
2*3 ≡ M23
≡ (λabc.a(bc))(λsz.s(sz))(λxy.x(x(xy)))
= (λc.(λsz.s(sz))((λxy.x(x(xy)))c)
= (λcz.((λxy.x(x(xy)))c)(((λxy.x(x(xy)))c)z))
= (λcz.(λy.c(c(cy))))(c(c(cz)))
= (λcz.c(c(c(c(cz)))))
≡ 6
```

Multiplying numbers in lambda calculus is pretty simple and straightforward. But, before you continue to more arithmetic functions, let's tackle first truth values and conditionals, which is a prerequisite in learning the other functions.

Truth, falsity, and friends

Booleans

The representations of true and false in lambda calculus, are succinct and elegant:

```
T ≡ (λxy.x)
F ≡ (λxy.y)
```

In action:

```
Tab ≡ (λxy.x)ab = a
Fab ≡ (λxy.y)ab = b
```

Logical operations

The three basic operators, And, Or, and Not:

```
∧ ≡ λxy.xy(λuv.v) ≡ λxy.xyF
∨ ≡ λxy.x(λuv.u)y ≡ λxy.xTy
¬ ≡ λx.x(λuv.v)(λab.a) ≡ λx.xFT
```

Let's see if **¬T** is indeed **F**:

```
¬T
≡ λx.x(λuv.v)(λab.a)(λcd.c)
≡ TFT
≡ (λcd.c)(λuv.v)(λab.a)
= (λuv.v)
≡ F
```

Let's count, backwards!

Predecessor

The predecessor of a number is defined as the preceding number determined when counting backwards. The reason why the discussion on the predecessor function is being done separately is that it isn't intuitively easy to determine at first, and that knowledge about other functions is important in understanding it.

Let's say you have a pair, something like (y, x), wherein the first element is one step above, or the successor the second element. Since the first element is the successor, that means the second element is the predecessor. Visually:

```
(z+1, z) = (z, z-1)
```

Therefore,

```
x = Py iff y = Sx
```

That is, **x** is the predecessor of **y**, if and only if, **y** is the successor of **x**.

So, to determine the predecessor of a number **x**, you create a pair like above, then select the second element.

Let's define some basic units. A pair looks like:

```
(λz.zab)
```

And the smallest unit of pair is:

```
(λz.z00) ≡ (λz.z(λsz.z)(λsz.z))
```

To select the first and second elements of a pair, you use **`T`** and **`F`**:

```
(λz.zab)(λxy.x) ≡ (λz.zab)T = Tab = a
(λz.zab)(λxy.y) ≡ (λz.zab)F = Fab = b
```

You need a function that takes a pair, then creates a new pair, wherein the first element is the successor of the second element.

```
Name: Q
Profile: (λpz.z(S(pT))(pT))
Inputs: (a, b)
Outputs: (S(a), b)
Usage: Q(a,b)
```

Let's test that out:

```
Q(λz.z00)
≡ (λpz.z(S(pT))(pT))(λz.z00)
= (λpz.z(S(pT))(pT))(λz.z(λsz.z)(λsz.z))
= (λz.z(λsz.s(z)(λsz.z)))
≡ (λz.z10)
```

Looks correct. You can now build your predecessor function:

```
Name: P
Profile: (λn.nQ(λz.z00))F
Inputs: N, where N is a natural number
Outputs: N-1
Usage: PN
```

Let's test that out:

```
P1
≡ ((λn.nQ(λz.z00))F)1
≡ ((λn.nQ(λz.z00))F)(λsz.s(z))
= (1Q(λz.z00))F
= ((λsz.s(z)((λpz.z((S(pT))(pT)))(λz.z(λsz.z)(λsz.z)))))(λxy.y)
= (λz.((λpz.z((S(pT))(pT)))(λz.z(λsz.z)(λsz.z)))z)(λxy.y)
≡ (λu.((λpz.z((S(pT))(pT)))(λz.z(λsz.z)(λsz.z)))u)(λxy.y)
= ((λpz.z((S(pT))(pT)))(λz.z(λsz.z)(λsz.z)))(λxy.y)
= (λz.z((S((λz.z(λsz.z)(λsz.z))T)((λz.z(λsz.z)(λsz.z))T))))(λxy.y)
= (λz.z((S(λsz.z)(λsz.z))))(λxy.y)
= (λz.z((λsz.s(z)(λsz.z))))(λxy.y)
= (λxy.y)((λsz.s(z)(λsz.z)))
= (λsz.z)
≡ 0
```

Subtraction

Now that you have the predecessor function, you can build your subtraction function.

```
B ≡ (λxy.yPx)
```

Let's test that out:

```
B11
≡ (λxy.yPx) (λsz.s(z)) (λsz.s(z))
= (λsz.s(z)) (P(λsz.s(z)))
= (λsz.s(z)) (λsz.z)
= (λz.(λsz.z)z)
= (λz.(λz.z))
≡ (λsz.z)
≡ 0
```

Closing remarks

You've just scratched the surface of lambda calculus, but you have just witnessed its immense expressive power, considering how minimal the system is defined. In our next article, we'll demystify even more lambda calculus magic. Stay tuned!

References

- http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf
- http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf
- http://palmstroem.blogspot.com/2012/05/lambda-calculus-for-absolute-dummies.html
- http://www.users.waitrose.com/~hindley/SomePapersPDFs/2006CarHin,HistlamRp.p

---