Ruby ♡ λ Calculus

Christopher Mark Gore

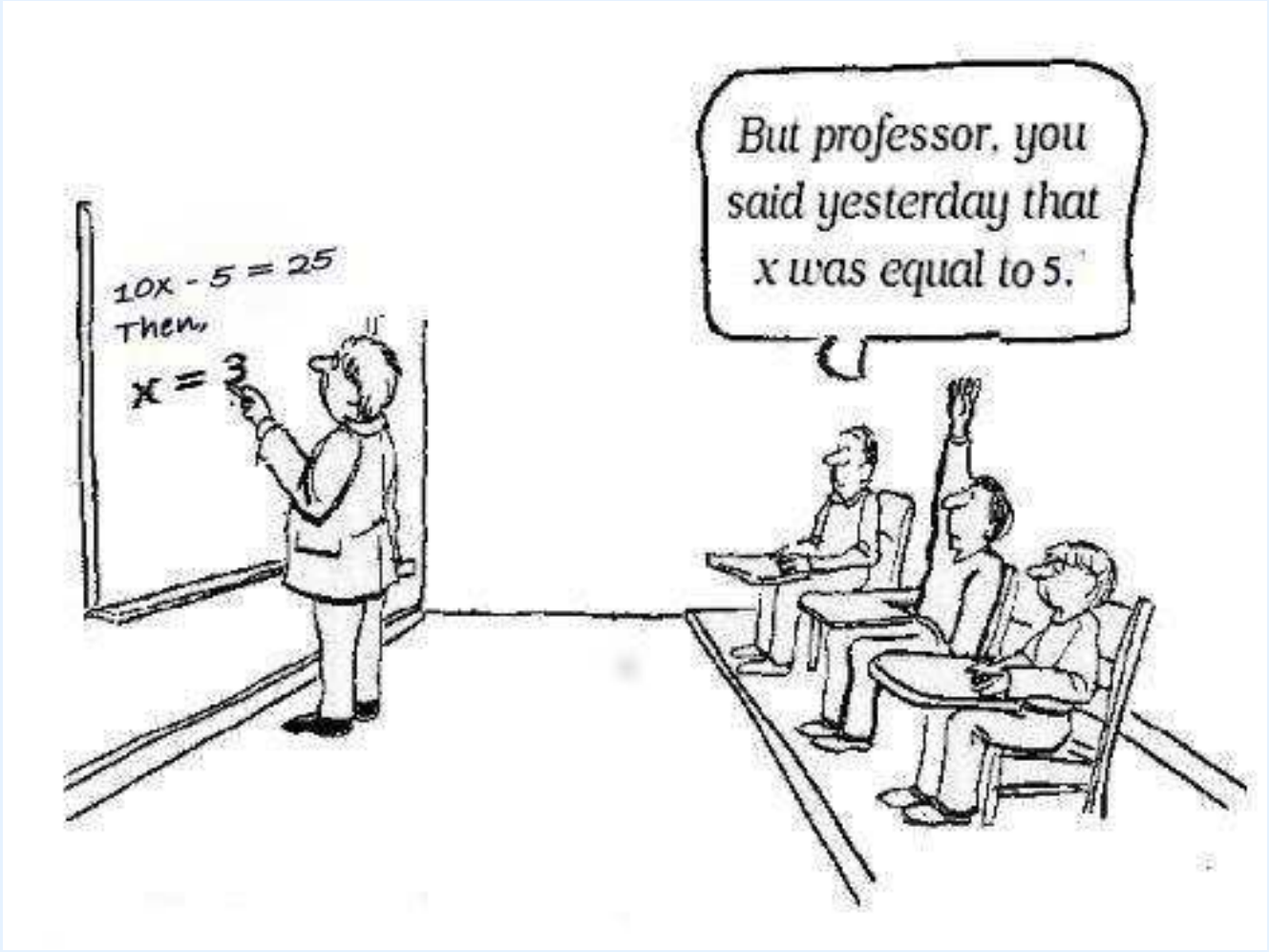
cgore.com

Monday, October 13, AD 2014

Math Can be Difficult, but Fun!

- Mathematics is about finding patterns.
- Mathematics is usually the simplest way to talk about complex subjects.
- Mathematics can be confusing because of:
  - bad notation,
  - bad textbooks,
  - bad teachers,
  - *and sometimes* …

…Bad Students



Computer Science is Mathematics

- Computer science **is not** computer programming.
- Computer science **is not** software engineering.
- Computer science **is not** science, it's better than that:
- Computer science **is actually** mathematics.
- Computer science **really is** useful for computer programmers, in spite of some arguments to the contrary.

Lambda Calculus

- Lambda calculus **is not** the same "calculus" you learned in your high school or college "Calculus" class.
- It's a different calculus: there are a lot of others than just these two (and a lot of other algebras than the normal one as well.)
- It is actually **easier to use** for specific tasks than other options (typically making mathematical proofs about algorithms.)

Alonzo Church Invented It

"Lambda's where it's at."

All of Lambda Calculus on Just One Slide!

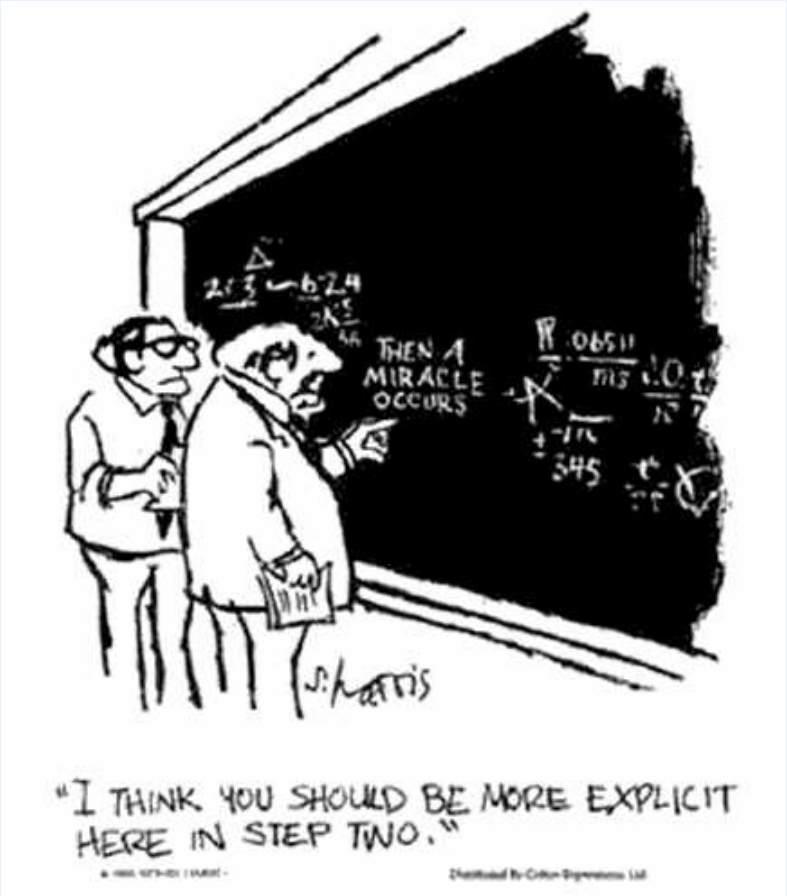
- Where e is any expression,
- f is any function,
- n is any name, and
- a is any application:

$$e := n|f|a$$

$$f := \lambda n.e$$

$$a := ee$$

We're Done!



Actually, some explanation might be in order …

Cool Facts about Lambda Calculus

- It's the smallest universal programming language in the universe!
- Church developed the lambda calculus to formalize the concept of effective computability in the 1930's.
- It's Turing complete!
- There is only one transformation rule: variable substitution.

Names

- Names (or *variables*, or *atoms*) are just any unique identifier, typically just a single letter.
- We'll mostly just use single letters: a, b, c, etc.
- The best equivalent in Ruby are symbols:
  - `:a`,
  - `:b`,
  - `:c`,
  - `:joe_don_baker`,
  - etc.

Expressions

The most important part of lambda calculus is the concept of an expression.

$$e := n|f|a$$

From this we see it's just any name, function, or application. Expressions are **everything!**

Functions

- A function[a] (also called an *abstraction*) just maps from a name to an expression: $f := \lambda n.e$
- The name on the left is the *argument* for the function, and the expression to the right is the *body* of the function.
- Functions have most of the syntax in the entire language: the lambda (λ) and the period (.).
- The closest equivalent in Ruby is, of course, the function. But, only taking in a single argument.

```
->(n){e}
```

[a] Technically, this must be a *computable function*, but don't worry about what that actually means.

Applications

An application is just two expressions. They are really here just for glue, so that expressions can expand out like a tree forever, don't worry about them.

$$a := ee$$

But, we *apply* functions to expressions, that is the most common direct use of them.

Parentheses and Left Association

Expressions can be surrounded by parentheses for clarity of order.

$$(E) = E$$

For *clarity*[a] the left-association rule is used:

$$E_1 E_2 E_3 = (E_1 E_2) E_3$$

$$E_1 E_2 E_3 E_4 = ((E_1 E_2) E_3)\, E_4$$

$$E_1 E_2 E_3 \ldots E_n = (\ldots((E_1 E_2) E_3) \ldots E_n)$$

[a] Actually, this is the main problem with the lack of clarity found in the Lambda Calculus. It allows for quite terse and short formulas, but not really clear. It's kind of like RPN, it just takes some getting used to.

The Identity Function

The identity function:

$$\lambda x.x$$

It just returns what it is given.

How would this look in Ruby?

```
->(x){x}
```

Applying Functions

We can *apply* a function to an expression: Let's apply the identity function to a really simple expression, just a single atom:

$$(\lambda x.x)y = [y/x]x = y$$

The *bracket notation*[a] that we see in $[y/x]$ tells us to replace all x's with y's in the expression that follows.

How would this look in Ruby?

```
->(x){x}[:y] # Returns :y
```

We can easily see what is happening here: where ever there is an x in the body of the function, the right hand side of the application is substituted, in this case y.

[a] Sometimes to confuse you authors will write this as $[x := y]$ instead; notice that they are backwards, $[x := y] \equiv [y/x]$.

Free and Bound Variables

All variables are local to the scope of their function.

$$\lambda x.xy$$

In this function definition:

- x is a *bound* variable.
- y is a *free* variable.

But this goes *per expression*, so in

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

we have x as first bound and then free; the two instances of x are completely independent of each other.

Two Irritations That Are Actually Intentional Features

- Functions are always anonymous: we get around this by *unofficially* assigning names to expressions, like

$$\boldsymbol{I} = \lambda x.x$$

- Functions only ever take a single variable: we get around this with *currying*, which we'll discuss shortly.

This is to make the semantics more simple, but can be irritating sometimes otherwise.

α Equivalence

The specific symbols used for the arguments of function definitions don't matter, and can be switched out at will.

$$\boldsymbol{I} = \lambda x.x \equiv \lambda y.y \equiv \lambda z.z \equiv \lambda ☺.☺$$

β Reductions

The formal name for using those brackets is a β *reduction*. In a class setting, you might get the limitless joy of working a lot of these out by hand for hours on end.

Here's a short one I found on the Internet:[a]

[a] `https://gist.github.com/homelinen/4012463`

$$
\begin{array}{ll}
 & (\lambda xyz.xyz)(\lambda x.xx)(\lambda x.x)x \\
\to_\beta & (\lambda yz.xyz)[x := \lambda x.xx](\lambda x.x)x \\
\equiv & (\lambda yz.(\lambda x.xx)yz)(\lambda x.x)x \\
\to_\beta & (\lambda yz.(xx)[x := y]z)(\lambda x.x)x \\
\equiv & (\lambda yz.yyz)(\lambda x.x)x \\
\to_\beta & (\lambda z.yy)[y := \lambda x.x]x \\
\equiv & (\lambda z.(\lambda x.x)(\lambda x.x)z)x \\
\to_\beta & (\lambda z.x[x := \lambda x.x]z)x \\
\equiv & (\lambda z.(\lambda x.x)z)x \\
\to_\beta & (\lambda z.x[x := z])x \\
\equiv & (\lambda z.z)x \\
\to_\beta & z[z := x] \\
\equiv & x
\end{array}
$$

Currying

One of the more irritating things about functions in lambda calculus is that they can take only one argument. What do we do when we want more arguments? We curry.

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

Currying in Ruby

The equivalent in Ruby is something like this.

```
# Like this, the simple version.
i1 = ->(x){->(y){y}[x]}
i1[:howza] # Returns :howza
# Or like this (since Ruby 1.9.)
i2 = ->(x){ ->(y){y}.curry[x] }
i2[:howza] # Also returns :howza
```

Questions?

References

- *Introduction to Lambda Calculus* (Henk Barendregt and Erik Barendsen, 2000.)
- *The Untyped Lambda Calculus* (Manuel Eberl, 2011.)
- *A Tutorial Introduction to the Lambda Calculus* (Raúl Rojas, 1997.)
- λ *Calculus – Church Numerals and Lists* (Michael Smith, 2004.)