Lambda Calculus Quick Start

Brady Dean
11/19/18

Lambda Calculus is a mathematical model of computation created by Alonzo Church in the 1930s. It uses a simple syntax and two procedures to describe all computable things. As you will see, Lambda Calculus is very different from context-free grammars, but I will define Lambda Calculus using a context-free grammar to avoid the mathematical jargon. Lambda Calculus has three *terms*.

$$< variable > ::= a, b, c, etc$$
$$< abstraction > ::= \lambda < variables > . < term >$$
$$< application > ::= < term >< term >$$

A Lambda Calculus expression is any sentence in this grammar. You may place parenthesis around abstractions and applications to disambiguate. Abstraction is synonymous with *function*.

Given an abstraction in the form $\lambda f . E$

- $\lambda f .$ is the *head*.
- E is the *body*.

The head may declare multiple variables and you will see later that the evaluation algorithm uses *currying* to evaluate each argument in order.

Here are some examples:

- Variable: x
- Abstraction: $\lambda x . x + 2$
- Multiple parameters: $\lambda xy . x + y$
- Application: $(\lambda x . x + 2)\ y$

Abstractions have *bound* and *free* variables.

- Bound variables appear in the head.
- Free variables do not.
- E.g. In $\lambda x.xx$ and $\lambda x.xy$, x is bound and y is free.

You will need to know about *alpha equivalence*. Two abstractions are alpha equivalent if the first equals the second after changing only bound variable names. For example:

$$\lambda x.x = \lambda y.y$$
$$\lambda x.xy = \lambda z.zy$$

Any variable that appears between the lambda and dot is bound to the abstraction and as such, you may change its name to avoid ambiguous terms during evaluation.

$$(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x'.x'x')$$

The key to understanding lambda calculus is the realization that *everything* is a function. Numbers are function, variables are functions, operators are functions, Booleans, etc.

You may define numbers by successive applications. Zero is zero applications of s. One is one application of s. Two is two applications of s, etc. These numbers are Church Numerals. In practice, you may write the numbers and operators as symbols instead of abstractions. Some examples:

$$0 \coloneqq \lambda s.\lambda z.z$$
$$1 \coloneqq \lambda s.\lambda z.sz$$
$$2 \coloneqq \lambda s.\lambda z.s(sz)$$
$$3 \coloneqq \lambda s.\lambda z.s\big(s(sz)\big)$$
$$n \coloneqq \cdots etc$$
$$m + n \coloneqq \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$
$$m * n \coloneqq \lambda m.\lambda n.\lambda f.\lambda x.m(nf)x$$

To begin evaluating expressions I will define *normal form* to describe an expression that has no more arguments to apply. Such an expression has no application terms remaining. *Beta reduction* is an algorithm that evaluates an expression into its normal form. It is analogous to function application.

1. Take an application in the form $(\lambda f.E)M$
2. Perform alpha conversion if a bound variable in M appears in the left term.
3. Substitute all occurrences of f in E with M
4. Remove the head, λf.
5. If another application remains, go to step 1.

Given a compound expression, first *uncurry* the abstractions by expanding the heads. E.g., $\lambda xy.x + y \rightarrow \lambda x.\lambda y.x + y$. Then you pick the leftmost, outermost application term to reduce first. Currying means to apply one argument to the abstraction at a time, which naturally happens with normal order evaluation. Notice that normal order application is left associative. When no more application terms remain, the result is the normal form of the original expression. Normal order is the default ordering for Lambda Calculus, but you may also use Applicative ordering (LISP), or Call by Need ("lazy evaluation" in Haskell). As an interesting tidbit, normal order evaluation is parallelizable and Haskell takes advantage of this property in some computations.

Evaluate:

$$(\lambda xyz.\, xyz)(\lambda x.\, xx)(\lambda x.\, x)x$$
$$(\lambda x.\, \lambda y.\, \lambda z.\, xyz)(\lambda x.\, xx)(\lambda x.\, x)x$$
$$(\lambda x.\, \lambda y.\, \lambda z.\, xyz)(\lambda x'.\, x'x')(\lambda x.\, x)x$$
$$(\lambda y.\, \lambda z.\, (\lambda x'.\, x'x')yz)(\lambda x.\, x)x$$
$$(\lambda z.\, (\lambda x'.\, x'x')(\lambda x.\, x)z)x$$
$$(\lambda z.\, (\lambda x'.\, x'x')(\lambda x.\, x)z)x''$$
$$(\lambda x'.\, x'x')(\lambda x.\, x)x''$$
$$(\lambda x.\, x)(\lambda x.\, x)x''$$
$$(\lambda x.\, x)(\lambda x'.\, x')x''$$
$$(\lambda x'.\, x')x''$$
$$x''$$
$$x$$

While performing beta reduction, there are many applications of alpha conversion, and the reduce form of $(\lambda xyz.\, xyz)(\lambda x.\, xx)(\lambda x.\, x)x$ is x.

Evaluate: (alpha conversion not shown)

$$(\lambda x.\, xx)(x.\, xx)$$
$$(\lambda x.\, xx)(x.\, xx)$$
$$(\lambda x.\, xx)(x.\, xx)$$
$$(\lambda x.\, xx)(x.\, xx)$$
$$(\lambda x.\, xx)(x.\, xx)$$

...

This is a divergent expression; beta reduction will never find a normal form. However, this construct is used to define the Y-combinator, which allows for recursion in Lambda Calculus.

$$Y \coloneqq \lambda f.\big(\lambda x.f(xx)\big)\big(\lambda x.f(xx)\big)$$

Try applying an abstraction to it and you will see what I mean. For instance, $Y(\lambda x.x+1)$ creates an expression that repeatedly applies $\lambda x.x+1$ infinitely many times. It evaluates like this

$$(\lambda x.x+1)(\lambda x.x+1)(\lambda x.x+1)(\lambda x.x+1)\ ...$$

You may recall that recursion is just repeated application of a function to itself. If you sneak a base case in, then violá, you make a recursive expression.

Variables, abstractions, and recursion are the building blocks used to solve problems. I will not bore you with Church's formal proof, but you should have an intuition that Lambda Calculus will encode all computable problems.