Lambda Calculus

- Developed by Alonzo Church and Haskell Curry in the 1930s and 40s.
- Branch of mathematical logic. Provides a foundation for mathematics. Describes — like Turing machines — that which can be effectively computed.
- In contrast to Turing machines, lambda calculus does not care about any underlying “hardware” but rather uses simple syntactic transformation rules to define computations.

Lambda Calculus

- A theory of functions where functions are manipulated in a purely syntactic way.
- In lambda Calculus, everything is represented as a function.
- Functional programming languages are variations on lambda calculus.
- Lambda calculus is the theoretical foundation of functional programming languages.
- “the smallest universal programming language”.
- Sparse syntax and simple semantics — still, powerfull enough to represent all computable functions.

Introductory Example
Let's look at how a **lambda expression** is evaluated.

You are not expected to understand this, yet.

The function

\[ f(x, y, z) = x \cdot y + z \]

looks like this in lambda calculus:

\[ f \equiv (\lambda x. (\lambda y. (\lambda z. \text{add} (\text{mul} x y) z))) \]

Let's evaluate

\[ f(3, 4, 5) = 3 \cdot 4 + 5 \]

or, in Scheme

\[ > (((((\text{lambda} (x) (\text{lambda} (y) (\text{lambda} (z) (+ (* x y) z))) 3) 4) 5) 17 \]

or, in lambda calculus:

\[ (((((\lambda x. (\lambda y. (\lambda z. \text{add} (\text{mul} x y) z))) 3) 4) 5) 17 \]

Evaluation is done by substitution. The first step is to replace \( x \) with 3:

\[ (((((\lambda x. (\lambda y. (\lambda z. \text{add} (\text{mul} x y) z))) 3) 4) 5) \Rightarrow ((\lambda y. (\lambda z. \text{add} (\text{mul} 3 y) z)) 4) 5) \]

Next, we replace \( y \) with 4:

\[ (((((\lambda x. (\lambda y. (\lambda z. \text{add} (\text{mul} x y) z))) 3) 4) 5) \Rightarrow ((\lambda y. (\lambda z. \text{add} (\text{mul} 3 4) z)) 4) 5) \Rightarrow ((\lambda z. \text{add} (\text{mul} 12 z) 5) \]

Next, we multiply \( 3 \cdot 4 \):

\[ (((((\lambda x. (\lambda y. (\lambda z. \text{add} (\text{mul} x y) z))) 3) 4) 5) \Rightarrow ((\lambda y. (\lambda z. \text{add} (\text{mul} 3 y) z)) 4) 5) \Rightarrow ((\lambda z. \text{add} (\text{mul} 12 z) 5) \]

\[ (((\lambda z. \text{add} 12 z) 5) \]
Introductory Example...

Finally, we replace \( z \) by 5 and add:

\[
(((\lambda x. (\lambda y. (\lambda z. add (mul x y) z))) 3) 4) 5) \Rightarrow
\]

\[
(((\lambda y. (\lambda z. add (mul 3 y) z)) 4) 5) \Rightarrow
\]

\[
((\lambda z. add (mul 3 4) z) 5) \Rightarrow
\]

\[
((\lambda z. add 12 z) 5)
\]

(\( add \ 12 \ 5 \))

17

Syntax

There are four kinds of lambda expressions:

1. \textbf{variables} (lower-case letters)
2. \textbf{predefined constants and operations} (numbers and arithmetic operators)
3. \textbf{function applications}
4. \textbf{function abstraction} (function definitions)

expression ::= 

\[
\text{variable} \ |
\]

\[
\text{constant} \ |
\]

\[
(\text{expression expression}) \ |
\]

\[
(\lambda \text{variable . expression})
\]

Syntax — Function Application

In the expression

\[
(E_1 \ E_2)
\]

we expect \( E_1 \) to evaluate to a function, either a predefined one like \texttt{add} or \texttt{mul} or one defined by ourselves, as a lambda abstraction.

For example, in

\[
(\text{sqrt} \ 9)
\]

\texttt{sqrt} represents the constant (predefined) square root function, and 9 it’s argument.
Most authors leave out parentheses whenever possible. We will assume function application associates left-to-right.

Example:

\[ f A B \]

should be interpreted as

\[ ((f A) B) \]

not

\[ (f (A B)) \]

In

\[ (\lambda x. \text{times } x x) \]

the \( \lambda \) introduces \( x \) as a formal parameter to the function definition.

Function application binds tighter than function definition. For example,

\[ (\lambda x. A B) \]

should be interpreted as

\[ (\lambda x.(A B)) \]

not

\[ ((\lambda x.A) B) \]

In other words, the scope of

\[ (\lambda x. \cdots) \]

extends as far right as possible.

For example,

\[ (\lambda x. A B C) \]

means

\[ (\lambda x.((A B) C)) \]

not

\[ ((\lambda x.(A B)) C) \]

or

\[ ((\lambda x.A) (B C)) \]

In

\[ (\lambda x.E) \]

the variable \( x \) is said to be **bound** within \( E \).

This is similar to **scope** in other programming languages:

\[
\{ 
\quad \text{int } x;; 
\quad \ldots 
\quad \text{print } x 
\}
\]
Variables...

- In

\((\lambda x.\text{square } y)\)

the variable \(y\) is said to be **free**.

- Similar to other programming languages, a free variable is typically bound within an outer scope, like \(y\) here:

```cpp
{ int y;
  ...
  print y
}
```

Variables can hold any kind of value, including functions. We say functions are **Polymorphic** — they can take arguments of any type.

Syntax — Naming expressions

- We can give expressions names, so we can refer to them later:

\[
\text{square} \equiv (\lambda x.(\text{times } x x))
\]

\(\equiv\) means **is an abbreviation for**.

Syntax — Multiple Arguments

- A lambda abstraction can only take one argument:

\[(\lambda x.(\text{times } x x))\]

- To simulate multi-argument functions we use **currying**.
- The abstraction

\[
(\lambda f.(\lambda x.f(f x)))
\]

represents a function with two arguments, a function \(f\), and a value \(x\), and which applies \(f\) twice to \(x\).
Syntax — Multiple Arguments...

Example:

\[ (((\lambda f. (\lambda x.f (f x))) \text{sqr}) \ 3) = \]
\[ (((\lambda x. \text{sqr} (\text{sqr} \ x)) \ 3) = \]
\[ \text{sqr} (\text{sqr} \ 3) = \]
\[ (\text{sqr} \ 9) = 81 \]

In the first step, \( f \) is replaced by \( \text{sqr} \) (the squaring function).
In the second step, \( x \) is replaced by 3.

Some authors use the abbreviation

\[ (\lambda x \ y \ z. E) \]

to mean

\[ (\lambda x. (\lambda y. (\lambda z. E))) \]

In general, different books on lambda calculus will use slight variations in syntax.

Example — The identity function

This

\[ (\lambda x. x) \]

is the identity function.

The expression

\[ ((\lambda x. x) \ E) \]

will return \( E \) for any lambda expression \( E \).
For example, the expression

\[ ((\lambda x. x) \ (\text{sqr} \ 3)) \]

will return 9.
The expression

$$(\lambda n.\text{add } n \ 1)$$

is the integer successor function.

So,

$$( (\lambda n.\text{add } n \ 1) \ 5)$$

would return 6.

Both \text{add} and 1 need to be predefined constants in the language. Later we will see how they can be defined in the calculus from first principles.

Consider the expression

$$(\lambda n.\lambda f.\lambda x. f(n\ f\ x))(\lambda g, \lambda y.\ g\ y)$$

Identify the lambda expressions, which extend as far to the right as possible:

$$(\lambda n.\lambda f.\lambda x. f(n\ f\ x))(\lambda g, \lambda y.\ g\ y) =$$

$$(\lambda n.\lambda f.\lambda x. f(n\ f\ x))(\lambda g, \lambda y.\ g\ y) =$$

$$(\lambda n.\lambda f.\lambda x. f(n\ f\ x))(\lambda g, \lambda y.\ g\ y) =$$

Next, group applications by associating them to the left:

$$(\lambda n.\lambda f.\lambda x. f(n\ f\ x))(\lambda g, \lambda y.\ g\ y) =$$

Finally, insert parenthesis:

$$(\lambda n.\lambda f.\lambda x. f(n\ f\ x))(\lambda g, \lambda y.\ g\ y) =$$
**Example — Bound/Free Variables**

Find the bound and free variables in the expression

\[ \lambda x. y \lambda y. y \ x \]

First, parenthesize:

\[ (\lambda x. (y \ (\lambda y. (y \ x)))) \]

- \( x \) is bound, \( y \) is free, \( y \) is bound:
  \[ (\lambda x. (y \ (\lambda y. (y \ x)))) \]

---

**Readings and References**

- Read pp. 614–615, in Scott.

---

**Acknowledgments**