Lambda calculus (or λ-calculus) was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s to describe functions in an unambiguous and compact manner. Many real languages are based on the lambda calculus, including Lisp, Scheme, Haskell, and ML. A key characteristic of these languages is that functions are values, just like integers and booleans are values: functions can be used as arguments to functions, and can be returned from functions.

The name "lambda calculus" comes from the use of the Greek letter lambda (λ) in function definitions. (The letter lambda has no significance.) "Calculus" means a method of calculating by the symbolic manipulation of expressions.

Intuitively, a function is a rule for determining a value from an argument. Some examples of functions in mathematics are

$$f(x) = x^3$$
$$g(y) = y^3 - 2y^2 + 5y - 6.$$

1 Syntax

The pure λ-calculus contains just function definitions (called *abstractions*), variables, and function *application* (i.e., applying a function to an argument). If we add additional data types and operations (such as integers and addition), we have an *applied* λ-calculus. In the following text, we will sometimes assume that we have integers and addition in order to give more intuitive examples.

The syntax of the pure λ-calculus is defined as follows.

$$
\begin{array}{lll}
e ::= & x & \text{variable} \\
& | \ \lambda x.\, e & \text{abstraction} \\
& | \ e_1 \ e_2 & \text{application}
\end{array}
$$

An abstraction $\lambda x.\, e$ is a function: variable x is the *argument*, and expression e is the *body* of the function. Note that the function $\lambda x.\, e$ doesn't have a name. Assuming we have integers and arithmetic operations, the expression $\lambda x.\, x^2$ is a function that takes an argument x and returns the square of x.

An application $e_1 \ e_2$ requires that e_1 is (or evaluates to) a function, and then applies the function to the expression e_2. For example, $(\lambda x.\, x^2) \ 5$ is, intuitively, equal to 25, the result of applying the squaring function $\lambda x.\, x^2$ to 5.

Here are some examples of lambda calculus expressions.

$\lambda x.\, x$	a lambda abstraction called the *identity function*
$\lambda x.\, (f\ (g\ x)))$	another abstraction
$(\lambda x.\, x)\ 42$	an application
$\lambda y.\, \lambda x.\, x$	an abstraction that ignores its argument and returns the identity function

Lambda expressions extend as far to the right as possible. For example $\lambda x.\, x\,\lambda y.\, y$ is the same as $\lambda x.\, x\,(\lambda y.\, y)$, and is not the same as $(\lambda x.\, x)\,(\lambda y.\, y)$. Application is left associative. For example $e_1\, e_2\, e_3$ is the same as $(e_1\, e_2)\, e_3$. In general, use parentheses to make the parsing of a lambda expression clear if you are in doubt.

1.1 Variable binding and α-equivalence

An occurrence of a variable in an expression is either *bound* or *free*. An occurrence of a variable x in a term is bound if there is an enclosing $\lambda x.\, e$; otherwise, it is *free*. A *closed term* is one in which all identifiers are bound.

Consider the following term:

$$\lambda x.\, (x\,(\lambda y.\, y\, a)\, x)\, y$$

Both occurrences of x are bound, the first occurrence of y is bound, the a is free, and the last y is also free, since it is outside the scope of the λy.

If a program has some variables that are free, then you do not have a complete program as you do not know what to do with the free variables. Hence, a well formed program in lambda calculus is a closed term.

The symbol λ is a *binding operator*, as it binds a variable within some scope (i.e., some part of the expression): variable x is bound in e in the expression $\lambda x.\, e$.

The name of bound variables is not important. Consider the mathematical integrals $\int_0^7 x^2 dx$ and $\int_0^7 y^2 dy$. They describe the same integral, even though one uses variable x and the other uses variable y in their definition. The meaning of these integrals is the same: the bound variable is just a placeholder. In the same way, we can change the name of bound variables without changing the meaning of functions. Thus $\lambda x.\, x$ is the same function as $\lambda y.\, y$. Expressions e_1 and e_2 that differ only in the name of bound variables are called *α-equivalent*, sometimes written $e_1 =_\alpha e_2$.

1.2 Higher-order functions

In lambda calculus, functions are values: functions can take functions as arguments and return functions as results. In the pure lambda calculus, every value is a function, and every result is a function!

For example, the following function takes a function f as an argument, and applies it to the value 42.

$$\lambda f.\, f\, 42$$

This function takes an argument v and returns a function that applies its own argument (a function) to v.

$$\lambda v.\, \lambda f.\, (f\, v)$$

2 Semantics

2.1 β-equivalence

Application $(\lambda x.\, e_1)\, e_2$ applies the function $\lambda x.\, e_1$ to e_2. In some ways, we would like to regard the expression $(\lambda x.\, e_1)\, e_2$ as equivalent to the expression e_1 where every (free) occurrence of x is replaced with e_2. For example, we would like to regard $(\lambda x.\, x^2)\, 5$ as equivalent to 5^2.

We write $e_1\{e_2/x\}$ to mean expression e_1 with all free occurrences of x replaced with e_2. There are several different notations to express this substitution, including $[x \mapsto e_2]e_1$ (used by Pierce), $[e_2/x]e_1$ (used by Mitchell), and $e_1[e_2/x]$ (used by Winskel).

Using our notation, we would like expressions $(\lambda x.\, e_1)\, e_2$ and $e_1\{e_2/x\}$ to be equivalent.

We call this equivalence, between $(\lambda x.\, e_1)\, e_2$ and $e_1\{e_2/x\}$, is called *β-equivalence*. Rewriting $(\lambda x.\, e_1)\, e_2$ into $e_1\{e_2/x\}$ is called a *β-reduction*. Given a lambda calculus expression, we may, in general, be able to perform β-reductions. This corresponds to executing a lambda calculus expression.

There may be more than one possible way to β-reduce an expression. Consider, for example, $(\lambda x.\, x + x)\, ((\lambda y.\, y)\, 5)$. We could use β-reduction to get either $((\lambda y.\, y)\, 5) + ((\lambda y.\, y)\, 5)$ or $(\lambda x.\, x + x)\, 5$. The order in which we perform β-reductions results in different semantics for the lambda calculus.

2.2 Call-by-value

Call-by-value (or CBV) semantics makes sure that functions are only called on values. That is, given an application $(\lambda x.\, e_1)\, e_2$, CBV semantics makes sure that e_2 is a value before calling the function.

So, what is a value? In the pure lambda calculus, any abstraction is a value. Remember, an abstraction $\lambda x.\, e$ is a function; in the pure lambda calculus, the only values are functions. In an applied lambda calculus with integers and arithmetic operations, values also include integers. Intuitively, a value is an expression that can not be reduced/executed/simplified any further.

We can give small step operational semantics for call-by-value execution of the lambda calculus. Here, v can be instantiated with any value (e.g., a function).

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e \to e'}{v\, e \to v\, e'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}$$

We can see from these rules that, given an application $e_1\, e_2$, we first evaluate e_1 until it is a value, then we evaluate e_2 until it is a value, and then we apply the function to the value—a β-reduction.

Let's consider some examples. (These examples use an applied lambda calculus that also includes reduction rules for arithmetic expressions.)

$$
\begin{aligned}
(\lambda x.\, \lambda y.\, y\, x)\, (5 + 2)\, \lambda x.\, x + 1 \ &\to (\lambda x.\, \lambda y.\, y\, x)\, 7\, \lambda x.\, x + 1 \\
&\to (\lambda y.\, y\, 7)\, \lambda x.\, x + 1 \\
&\to (\lambda x.\, x + 1)\, 7 \\
&\to 7 + 1 \\
&\to 8
\end{aligned}
$$

$$
\begin{aligned}
(\lambda f.\, f\, 7)\, ((\lambda x.\, x\, x)\, \lambda y.\, y) \ &\to (\lambda f.\, f\, 7)\, ((\lambda y.\, y)\, (\lambda y.\, y)) \\
&\to (\lambda f.\, f\, 7)\, (\lambda y.\, y) \\
&\to (\lambda y.\, y)\, 7 \\
&\to 7
\end{aligned}
$$

3

2.3   Call-by-name

Call-by-name (or CBN) semantics applies the function as soon as possible. The small step operational semantics are a little simpler, as they do not need to ensure that the expression to which a function is applied is a value.

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad\qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \to e_1\{e_2/x\}}$$

Let's consider the same examples we used for CBV.

$$
\begin{aligned}
(\lambda x.\, \lambda y.\, y\ x)\ (5+2)\ \lambda x.\, x+1\ &\to (\lambda y.\, y\ (5+2))\ \lambda x.\, x+1\\
&\to (\lambda x.\, x+1)\ (5+2)\\
&\to (5+2)+1\\
&\to 7+1\\
&\to 8
\end{aligned}
$$

$$
\begin{aligned}
(\lambda f.\, f\ 7)\ ((\lambda x.\, x\ x)\ \lambda y.\, y)\ &\to ((\lambda x.\, x\ x)\ \lambda y.\, y)\ 7\\
&\to ((\lambda y.\, y)\ (\lambda y.\, y))\ 7\\
&\to (\lambda y.\, y)\ 7\\
&\to 7
\end{aligned}
$$

Note that the answers are the same, but the order of evaluation is different. (Later we will see languages where the order of evaluation is important, and may result in different answers.)