Lambda calculus

Lambda calculus (also written as **λ-calculus**) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution. It is a universal model of computation that can be used to simulate any Turing machine and was first introduced by mathematician Alonzo Church in the 1930s as part of his research of the foundations of mathematics.

Lambda calculus consists of constructing lambda terms and performing reduction operations on them. In the simplest form of lambda calculus, terms are built using only the following rules:

Syntax	Name	Description
x	Variable	A character or string representing a parameter or mathematical/logical value
(λx.M)	Abstraction	Function definition (M is a lambda term). The variable x becomes bound in the expression.
(M N)	Application	Applying a function to an argument. M and N are lambda terms.

producing expressions such as: (λ*x*.λ*y*.(λz.(λ*x*.*z x*) (λ*y*.*z y*)) (*x y*)). Parentheses can be dropped if the expression is unambiguous. For some applications, terms for logical and mathematical constants and operations may be included.

The reduction operations include:

Operation	Name	Description
(λx.M[x]) → (λy.M[y])	α-conversion	Renaming the bound (formal) variables in the expression. Used to avoid name collisions.
((λx.M) E) → (M[x:=E])	β-reduction	Substituting the bound variable by the argument expression in the body of the abstraction

If repeated application of the reduction steps eventually terminates then by the Church-Rosser theorem it will produce a beta normal form.

Contents

Explanation and applications

Lambda calculus is Turing complete, that is, it is a universal model of computation that can be used to simulate any Turing machine.[1] Its namesake, the Greek letter lambda (λ), is used in **lambda expressions** and **lambda terms** to denote binding a variable in a function.

Lambda calculus may be *untyped* or *typed.* In typed lambda calculus, functions can be applied only if they are capable of accepting the given input's "type" of data. Typed lambda calculi are *weaker* than the untyped lambda calculus that is the primary subject of this article, in the sense that *typed lambda calculi can express less* than the untyped calculus can, but on the other hand typed lambda calculi allow more things to be proved; in the simply typed lambda calculusit is for example a theorem that every evaluation strategy terminates for every simply typed lambda-term, whereas evaluation of untyped lambda-terms need not terminate. One reason there are many different typed lambda calculi has been the desire to do more (of what the untyped calculus can do) without giving up on being able to prove strong theorems about the calculus.

Lambda calculus has applications in many different areas in mathematics, philosophy,[2] linguistics,[3][4] and computer science.[5] Lambda calculus has played an important role in the development of the theory of programming languages. Functional programming languages implement the lambda calculus. Lambda calculus also is a current research topic in category theory.[6]

History

The lambda calculus was introduced by mathematician Alonzo Church in the 1930s as part of an investigation into the foundations of mathematics.[7][8] The original system was shown to be logically inconsistent in 1935 when Stephen Kleene and J. B. Rosser developed the Kleene–Rosser paradox.[9][10]

Subsequently, in 1936 Church isolated and published just the portion relevant to computation, what is now called the untyped lambda calculus.[11] In 1940, he also introduced a computationally weaker, but logically consistent system, known as the simply typed lambda calculus.[12]

Until the 1960s when its relation to programming languages was clarified, the λ-calculus was only a formalism. Thanks to Richard Montague and other linguists' applications in the semantics of natural language, the λ-calculus has begun to enjoy a respectable place in linguistics[13] and computer science, too![14]

Informal description

Motivation

Computable functions are a fundamental concept within computer science and mathematics. The λ-calculus provides a simple semantics for computation, enabling properties of computation to be studied formally. The λ-calculus incorporates two simplifications that make this semantics simple. The first simplification is that the λ-calculus treats functions "anonymously", without giving them explicit names. For example, the function

$$\text{square\_sum}(x, y) = x^2 + y^2$$

can be rewritten in *anonymous form* as

$$(x, y) \mapsto x^2 + y^2$$

(read as "a tuple of x and y is mapped to $x^2 + y^2$"). Similarly,

$$\text{id}(x) = x$$

can be rewritten in anonymous form as

$$x \mapsto x$$

where the input is simply mapped to itself.

The second simplification is that the λ-calculus only uses functions of a single input. An ordinary function that requires two inputs, for instance the **square_sum** function, can be reworked into an equivalent function that accepts a single input, and as output returns *another* function, that in turn accepts a single input. For example,

$$(x, y) \mapsto x^2 + y^2$$

can be reworked into

$$x \mapsto (y \mapsto x^2 + y^2)$$

This method, known as currying, transforms a function that takes multiple arguments into a chain of functions each with a single argument.

Function application of the **square_sum** function to the arguments (5, 2), yields at once

$$
\begin{aligned}
&((x, y) \mapsto x^2 + y^2)(5, 2) \\
&= 5^2 + 2^2 \\
&= 29,
\end{aligned}
$$

whereas evaluation of the curried version requires one more step

$$
\begin{aligned}
&\Big((x \mapsto (y \mapsto x^2 + y^2))(5)\Big)(2) \\
&= (y \mapsto 5^2 + y^2)(2) \text{ // the definition of } x \text{ has been used with } 5 \text{ in the inner expression. This is like β-reduction.} \\
&= 5^2 + 2^2 \text{ // the definition of } y \text{ has been used with } 2. \text{ Again, similar to β-reduction.} \\
&= 29
\end{aligned}
$$

to arrive at the same result.

The lambda calculus

The lambda calculus consists of a language of **lambda terms**, which is defined by a certain formal syntax, and a set of transformation rules, which allow manipulation of the lambda terms. These transformation rules can be viewed as an equational theory or as an operational definition.

As described above, all functions in the lambda calculus are anonymous functions, having no names. They only accept one input variable, with currying used to implement functions with several variables.

Lambda terms

The syntax of the lambda calculus defines some expressions as valid lambda calculus expressions and some as invalid, just as some strings of characters are valid C programs and some are not. A valid lambda calculus expression is called a "lambda term".

The following three rules give an inductive definition that can be applied to build all syntactically valid lambda terms:

- a variable, x, is itself a valid lambda term
- if t is a lambda term, and x is a variable, then $(\lambda x.\, t)$ is a lambda term (called a **lambda abstraction**);
- if t and s are lambda terms, then (ts) is a lambda term (called an **application**).

Nothing else is a lambda term. Thus a lambda term is valid if and only if it can be obtained by repeated application of these three rules. However, some parentheses can be omitted according to certain rules. For example, the outermost parentheses are usually not written. See *Notation*, below.

A **lambda abstraction** $\lambda x.\, t$ is a definition of an anonymous function that is capable of taking a single input x and substituting it into the expression t. It thus defines an anonymous function that takes x and returns t. For example, $\lambda x.\, x^2 + 2$ is a lambda abstraction for the function $f(x) = x^2 + 2$ using the term $x^2 + 2$ for t. The definition of a function with a lambda abstraction merely "sets up" the function but does not invoke it. The abstraction binds the variable x in the term t.

An **application** ts represents the application of a function t to an input s, that is, it represents the act of calling function t on input s to produce $t(s)$.

There is no concept in lambda calculus of variable declaration. In a definition such as $\lambda x.\, x + y$ (i.e. $f(x) = x + y$), the lambda calculus treats y as a variable that is not yet defined. The lambda abstraction $\lambda x.\, x + y$ is syntactically valid, and represents a function that adds its input to the yet-unknown y.

Bracketing may be used and may be needed to disambiguate terms. For example, $\lambda x.\,((\lambda x.\,x)x)$ and $(\lambda x.\,(\lambda x.\,x))x$ denote different terms (although they coincidentally reduce to the same value). Here the first example defines a function that defines a function and returns the result of applying x to the child-function (apply function then return), while the second example defines a function that returns a function for any input and then returns it on application of x (return function then apply).

Functions that operate on functions

In lambda calculus, functions are taken to be first class values, so functions may be used as the inputs, or be returned as outputs from other functions.

For example, $\lambda x.\,x$ represents the identity function, $x \mapsto x$, and $(\lambda x.\,x)y$ represents the identity function applied to y. Further, $(\lambda x.\,y)$ represents the **constant function** $x \mapsto y$, the function that always returns y, no matter the input. In lambda calculus, function application is regarded as left-associative, so that stx means $(st)x$.

There are several notions of "equivalence" and "reduction" that allow lambda terms to be "reduced" to "equivalent" lambda terms.

Alpha equivalence

A basic form of equivalence, definable on lambda terms, is alpha equivalence. It captures the intuition that the particular choice of a bound variable, in a lambda abstraction, does not (usually) matter. For instance, $\lambda x.\,x$ and $\lambda y.\,y$ are alpha-equivalent lambda terms, and they both represent the same function (the identity function). The terms x and y are not alpha-equivalent, because they are not bound in a lambda abstraction. In many presentations, it is usual to identify alpha-equivalent lambda terms.

The following definitions are necessary in order to be able to define beta reduction:

Free variables

The **free variables** of a term are those variables not bound by a lambda abstraction. The set of free variables of an expression is defined inductively:

- The free variables of x are just x
- The set of free variables of $\lambda x.\,t$ is the set of free variables of t, but with x removed
- The set of free variables of ts is the union of the set of free variables of t and the set of free variables of s.

For example, the lambda term representing the identity $\lambda x.\,x$ has no free variables, but the function $\lambda x.\,yx$ has a single free variable, y.

Capture-avoiding substitutions

Suppose t, s and r are lambda terms and x and y are variables. The notation $t[x := r]$ indicates substitution of r for x in t in a *capture-avoiding* manner. This is defined so that:

- $x[x := r] = r$;
- $y[x := r] = y$ if $x \neq y$;
- $(ts)[x := r] = (t[x := r])(s[x := r])$;
- $(\lambda x.\,t)[x := r] = \lambda x.\,t$;
- $(\lambda y.\,t)[x := r] = \lambda y.\,(t[x := r])$ if $x \neq y$ and y is not in the free variables of r. The variable y is said to be "fresh" for r.

For example, $(\lambda x.\,x)[y := y] = \lambda x.\,(x[y := y]) = \lambda x.\,x$, and $((\lambda x.\,y)x)[x := y] = ((\lambda x.\,y)[x := y])(x[x := y]) = (\lambda x.\,y)y$.

The freshness condition (requiring that y is not in the free variables of r) is crucial in order to ensure that substitution does not change the meaning of functions. For example, a substitution is made that ignores the freshness condition: $(\lambda x.\, y)[y := x] = \lambda x.\, (y[y := x]) = \lambda x.\, x$. This substitution turns the constant function $\lambda x.\, y$ into the identity $\lambda x.\, x$ by substitution.

In general, failure to meet the freshness condition can be remedied by alpha-renaming with a suitable fresh variable. For example, switching back to our correct notion of substitution, in $(\lambda x.\, y)[y := x]$ the lambda abstraction can be renamed with a fresh variable z, to obtain $(\lambda z.\, y)[y := x] = \lambda z.\, (y[y := x]) = \lambda z.\, x$, and the meaning of the function is preserved by substitution.

Beta reduction

The beta reduction rule states that an application of the form $(\lambda x.\, t)s$ reduces to the term $t[x := s]$. The notation $(\lambda x.\, t)s \to t[x := s]$ is used to indicate that $(\lambda x.\, t)s$ beta reduces to $t[x := s]$. For example, for every s, $(\lambda x.\, x)s \to x[x := s] = s$. This demonstrates that $\lambda x.\, x$ really is the identity. Similarly, $(\lambda x.\, y)s \to y[x := s] = y$, which demonstrates that $\lambda x.\, y$ is a constant function.

The lambda calculus may be seen as an idealised version of a functional programming language, like Haskell or Standard ML. Under this view, beta reduction corresponds to a computational step. This step can be repeated by additional beta conversions until there are no more applications left to reduce. In the untyped lambda calculus, as presented here, this reduction process may not terminate. For instance, consider the term $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$. Here $(\lambda x.\, xx)(\lambda x.\, xx) \to (xx)[x := \lambda x.\, xx] = (x[x := \lambda x.\, xx])(x[x := \lambda x.\, xx]) = (\lambda x.\, xx)(\lambda x.\, xx)$. That is, the term reduces to itself in a single beta reduction, and therefore the reduction process will never terminate.

Another aspect of the untyped lambda calculus is that it does not distinguish between different kinds of data. For instance, it may be desirable to write a function that only operates on numbers. However, in the untyped lambda calculus, there is no way to prevent a function from being applied to truth values, strings, or other non-number objects.

Formal definition

Definition

Lambda expressions are composed of:

- variables v_1, v_2, ..., v_n, ...
- the abstraction symbols lambda 'λ' and dot '.'
- parentheses ( )

The set of lambda expressions, Λ, can be defined inductively:

1. If x is a variable, then $x \in \Lambda$
2. If x is a variable and $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$
3. If $M, N \in \Lambda$, then $(M\ N) \in \Lambda$

Instances of rule 2 are known as abstractions and instances of rule 3 are known as applications.[15]

Notation

To keep the notation of lambda expressions uncluttered, the following conventions are usually applied:

- Outermost parentheses are dropped: M N instead of (M N)
- Applications are assumed to be left associative: M N P may be written instead of ((M N) P).[16]
- The body of an abstraction extends as far right as possible: λx.M N means λx.(M N) and not (λx.M) N
- A sequence of abstractions is contracted: λx.λy.λz.N is abbreviated as λxyz.N[17][16]

Free and bound variables

The abstraction operator, λ, is said to bind its variable wherever it occurs in the body of the abstraction. Variables that fall within the scope of an abstraction are said to be *bound*. All other variables are called *free*. For example, in the following expression y is a bound variable and x is free: $\lambda y.x\ x\ y$. Also note that a variable is bound by its "nearest" abstraction. In the following example the single occurrence of x in the expression is bound by the second lambda: $\lambda x.y\ (\lambda x.z\ x)$

The set of *free variables* of a lambda expression, M, is denoted as FV(M) and is defined by recursion on the structure of the terms, as follows:

1. FV(x) = {x}, where x is a variable
2. FV(λx.M) = FV(M) \ {x}
3. FV(M N) = FV(M) ∪ FV(N)[18]

An expression that contains no free variables is said to be *closed*. Closed lambda expressions are also known as combinators and are equivalent to terms in combinatory logic.

Reduction

The meaning of lambda expressions is defined by how expressions can be reduced.[19]

There are three kinds of reduction:

- **α-conversion**: changing bound variables (**alpha**);
- **β-reduction**: applying functions to their arguments (**beta**);
- **η-conversion**: which captures a notion of extensionality (**eta**).

We also speak of the resulting equivalences: two expressions are *β-equivalent*, if they can be β-converted into the same expression, and α/η-equivalence are defined similarly

The term *redex*, short for *reducible expression*, refers to subterms that can be reduced by one of the reduction rules. For example, $(\lambda x.M)\ N$ is a beta-redex in expressing the substitution of N for x in M; if x is not free in M, $\lambda x.M\ x$ is also an eta-redex. The expression to which a redex reduces is called its reduct; using the previous example, the reducts of these expressions are respectively $M[x:=N]$ and M.

α-conversion

Alpha-conversion, sometimes known as alpha-renaming,[20] allows bound variable names to be changed. For example, alpha-conversion of $\lambda x.x$ might yield $\lambda y.y$. Terms that differ only by alpha-conversion are called *α-equivalent*. Frequently, in uses of lambda calculus, α-equivalent terms are considered to be equivalent.

The precise rules for alpha-conversion are not completely trivial. First, when alpha-converting an abstraction, the only variable occurrences that are renamed are those that are bound to the same abstraction. For example, an alpha-conversion of $\lambda x.\lambda x.x$ could result in $\lambda y.\lambda x.x$, but it could *not* result in $\lambda y.\lambda x.y$. The latter has a different meaning from the original. This is analogous to the programming notion of variable shadowing.

Second, alpha-conversion is not possible if it would result in a variable getting captured by a different abstraction. For example, if we replace x with y in $\lambda x.\lambda y.x$, we get $\lambda y.\lambda y.y$, which is not at all the same.

In programming languages with static scope, alpha-conversion can be used to make name resolution simpler by ensuring that no variable name masks a name in a containing scope (see alpha renaming to make name resolution trivial).

In the De Bruijn index notation, any two alpha-equivalent terms are syntactically identical.

Substitution

Substitution, written $E[V := R]$, is the process of replacing all free occurrences of the variable V in the expression E with expression R. Substitution on terms of the λ-calculus is defined by recursion on the structure of terms, as follows (note: x and y are only variables while M and N are any λ expression).

$$
\begin{aligned}
&x[x := N] && \equiv N \\
&y[x := N] && \equiv y, \text{ if } x \neq y \\
&(M_1\ M_2)[x := N] && \equiv (M_1[x := N])\ (M_2[x := N]) \\
&(\lambda x.M)[x := N] && \equiv \lambda x.M \\
&(\lambda y.M)[x := N] && \equiv \lambda y.(M[x := N]), \text{ if } x \neq y, \text{ provided } y \notin \mathrm{FV}(N)
\end{aligned}
$$

To substitute into a lambda abstraction, it is sometimes necessary to α-convert the expression. For example, it is not correct for $(\lambda x.y)[y := x]$ to result in $(\lambda x.x)$, because the substituted x was supposed to be free but ended up being bound. The correct substitution in this case is $(\lambda z.x)$, up to α-equivalence. Notice that substitution is defined uniquely up to α-equivalence.

β-reduction

Beta-reduction captures the idea of function application. Beta-reduction is defined in terms of substitution: the beta-reduction of $((\lambda V.E)\ E')$ is $E[V := E']$.

For example, assuming some encoding of $2, 7, \times$, we have the following β-reduction: $((\lambda n.n \times 2)\ 7) \to 7 \times 2$.

η-conversion

Eta-conversion expresses the idea of extensionality, which in this context is that two functions are the same if and only if they give the same result for all arguments. Eta-conversion converts between $\lambda x.(f\ x)$ and f whenever x does not appear free in f.

Normal forms and confluence

For the untyped lambda calculus, β-reduction as a rewriting rule is neither strongly normalising nor weakly normalising.

However, it can be shown that β-reduction is confluent. (Of course, we are working up to α-conversion, i.e. we consider two normal forms to be equal, if it is possible to α-convert one into the other).

Therefore, both strongly normalising terms and weakly normalising terms have a unique normal form. For strongly normalising terms, any reduction strategy is guaranteed to yield the normal form, whereas for weakly normalising terms, some reduction strategies may fail to find it.

Encoding datatypes

The basic lambda calculus may be used to model booleans, arithmetic, data structures and recursion, as illustrated in the following sub-sections.

Arithmetic in lambda calculus

There are several possible ways to define the natural numbers in lambda calculus, but by far the most common are the Church numerals, which can be defined as follows:

$$
\begin{aligned}
0 &:= \lambda f.\lambda x.x \\
1 &:= \lambda f.\lambda x.f\ x \\
2 &:= \lambda f.\lambda x.f\ (f\ x) \\
3 &:= \lambda f.\lambda x.f\ (f\ (f\ x))
\end{aligned}
$$

and so on. Or using the alternative syntax presented above in *Notation*:

```
0 := λfx.x
1 := λfx.f x
2 := λfx.f (f x)
3 := λfx.f (f (f x))
```

A Church numeral is a higher-order function—it takes a single-argument function f, and returns another single-argument function. The Church numeral n is a function that takes a function f as argument and returns the n-th composition of f, i.e. the function f composed with itself n times. This is denoted $f^{(n)}$ and is in fact the n-th power of f (considered as an operator); $f^{(0)}$ is defined to be the identity function. Such repeated compositions (of a single function f) obey the laws of exponents, which is why these numerals can be used for arithmetic. (In Church's original lambda calculus, the formal parameter of a lambda expression was required to occur at least once in the function body, which made the above definition of `0` impossible.)

One way of thinking about the Church numeral n, which is often useful when analysing programs, is as an instruction 'repeat n times'. For example, using the `PAIR` and `NIL` functions defined below, one can define a function that constructs a (linked) list of n elements all equal to x by repeating 'prepend another x element' n times, starting from an empty list. The lambda term is

```
λn.λx.n (PAIR x) NIL
```

By varying what is being repeated, and varying what argument that function being repeated is applied to, a great many different effects can be achieved.

We can define a successor function, which takes a Church numeral n and returns $n + 1$ by adding another application of f, where '(mf)x' means the function 'f' is applied 'm' times on 'x':

```
SUCC := λn.λf.λx.f (n f x)
```

Because the m-th composition of f composed with the n-th composition of f gives the $m+n$-th composition of f, addition can be defined as follows:

```
PLUS := λm.λn.λf.λx.m f (n f x)
```

`PLUS` can be thought of as a function taking two natural numbers as arguments and returning a natural number; it can be verified that

```
PLUS 2 3
```

and

```
5
```

are β-equivalent lambda expressions. Since adding m to a number n can be accomplished by adding 1 m times, an alternative definition is:

```
PLUS := λm.λn.m SUCC n
```
[21]

Similarly, multiplication can be defined as

```
MULT := λm.λn.λf.m (n f)
```
[17]

Alternatively

```
MULT := λm.λn.m (PLUS n) 0
```

since multiplying m and n is the same as repeating the add n function m times and then applying it to zero. Exponentiation has a rather simple rendering in Church numerals, namely

```
POW := λb.λe.e b
```

The predecessor function defined by `PRED n = n − 1` for a positive integer *n* and `PRED 0 = 0` is considerably more difficult. The formula

```
PRED := λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
```

can be validated by showing inductively that if *T* denotes `(λg.λh.h (g f))`, then $T^{(n)}(\lambda u.x) = (\lambda h.h(f^{(n-1)}(x)))$ for `n > 0`. Two other definitions of `PRED` are given below, one using conditionals and the other using pairs. With the predecessor function, subtraction is straightforward. Defining

```
SUB := λm.λn.n PRED m,
```

`SUB m n` yields `m − n` when `m > n` and `0` otherwise.

Logic and predicates

By convention, the following two definitions (known as Church booleans) are used for the boolean values `TRUE` and `FALSE`:

```
TRUE := λx.λy.x
FALSE := λx.λy.y
```

(Note that `FALSE` is equivalent to the Church numeral zero defined above)

Then, with these two λ-terms, we can define some logic operators (these are just possible formulations; other expressions are equally correct):

```
AND := λp.λq.p q p
OR := λp.λq.p p q
NOT := λp.p FALSE TRUE
IFTHENELSE := λp.λa.λb.p a b
```

We are now able to compute some logic functions, for example:

```
AND TRUE FALSE

    ≡ (λp.λq.p q p) TRUE FALSE →β TRUE FALSE TRUE
    ≡ (λx.λy.x) FALSE TRUE →β FALSE
```

and we see that `AND TRUE FALSE` is equivalent to `FALSE`.

A *predicate* is a function that returns a boolean value. The most fundamental predicate is `ISZERO`, which returns `TRUE` if its argument is the Church numeral `0`, and `FALSE` if its argument is any other Church numeral:

```
ISZERO := λn.n (λx.FALSE) TRUE
```

The following predicate tests whether the first argument is less-than-or-equal-to the second:

```
LEQ := λm.λn.ISZERO (SUB m n),
```

and since `m = n`, if `LEQ m n` and `LEQ n m`, it is straightforward to build a predicate for numerical equality

The availability of predicates and the above definition of `TRUE` and `FALSE` make it convenient to write "if-then-else" expressions in lambda calculus. For example, the predecessor function can be defined as:

```
PRED := λn.n (λg.λk.ISZERO (g 1) k (PLUS (g k) 1)) (λv.0) 0
```

which can be verified by showing inductively that `n (λg.λk.ISZERO (g 1) k (PLUS (g k) 1)) (λv.0)` is the add $n - 1$ function for $n > 0$.

Pairs

A pair (2-tuple) can be defined in terms of `TRUE` and `FALSE`, by using the Church encoding for pairs. For example, `PAIR` encapsulates the pair (x,y), `FIRST` returns the first element of the pair, and `SECOND` returns the second.

```
PAIR := λx.λy.λf.f x y
FIRST := λp.p TRUE
SECOND := λp.p FALSE
NIL := λx.TRUE
NULL := λp.p (λx.λy.FALSE)
```

A linked list can be defined as either NIL for the empty list, or the `PAIR` of an element and a smaller list. The predicate `NULL` tests for the value `NIL`. (Alternatively, with `NIL := FALSE`, the construct `l (λh.λt.λz.deal_with_head_h_and_tail_t) (deal_with_nil)` obviates the need for an explicit NULL test).

As an example of the use of pairs, the shift-and-increment function that maps `(m, n)` to `(n, n + 1)` can be defined as

```
Φ := λx.PAIR (SECOND x) (SUCC (SECOND x))
```

which allows us to give perhaps the most transparent version of the predecessor function:

```
PRED := λn.FIRST (n Φ (PAIR 0 0)).
```

Additional programming techniques

There is a considerable body of programming idioms for lambda calculus. Many of these were originally developed in the context of using lambda calculus as a foundation for programming language semantics, effectively using lambda calculus as a low-level programming language. Because several programming languages include the lambda calculus (or something very similar) as a fragment, these techniques also see use in practical programming, but may then be perceived as obscure or foreign.

Named constants

In lambda calculus, a library (computing) would take the form of a collection of previously defined functions, which as lambda-terms are merely particular constants. The pure lambda calculus does not have a concept of named constants since all atomic lambda-terms are variables, but one can emulate having named constants by setting aside a variable as the name of the constant, using lambda-abstraction to bind that variable in the main body, and apply that lambda-abstraction to the intended definition. Thus to use f to mean M (some explicit lambda-term) in N (another lambda-term, the "main program"), one can say

$$(\lambda f.N)\ M$$

Authors often introduce syntactic sugar, such as `let`, to permit writing the above in the more intuitive order

$$\texttt{let}\ f = M\ \texttt{in}\ N$$

By chaining such definitions, one can write a lambda calculus "program" as zero or more function definitions, followed by one lambda-term using those functions that constitutes the main body of the program.

A notable restriction of this `let` is that the name f is not defined in M, since M is outside the scope of the lambda-abstraction binding f; this means a recursive function definition cannot be used as the M with `let`. The more advanced `letrec` syntactic sugar construction that allows writing recursive function definitions in that naive style instead additionally employs fixed-point

combinators.

Recursion and fixed points

Recursion is the definition of a function using the function itself. Lambda calculus cannot express this as directly as some other notations: all functions are anonymous in lambda calculus, so we can't refer to a value which is yet to be defined, inside the lambda term defining that same value. However, recursion can still be achieved by arranging for a lambda expression to receive itself as its argument value, for example in `(λx.x x) E`.

Consider the factorial function `F(n)` recursively defined by

```
F(n) = 1, if n = 0; else n × F(n − 1).
```

In the lambda expression which is to represent this function, a *parameter* (typically the first one) will be assumed to receive the lambda expression itself as its value, so that calling it – applying it to an argument – will amount to recursion. Thus to achieve recursion, the intended-as-self-referencing argument (called `r` here) must always be passed to itself within the function body, at a call point:

```
G := λr. λn.(1, if n = 0; else n × (r r (n−1)))
```

with `r r x = F x = G r x` to hold, so `r = G` and

```
F := G G = (λx.x x) G
```

The self-application achieves replication here, passing the function's lambda expression on to the next invocation as an argument value, making it available to be referenced and called there.

This solves it but requires re-writing each recursive call as self-application. We would like to have a generic solution, without a need for any re-writes:

```
G := λr. λn.(1, if n = 0; else n × (r (n−1)))
```

with `r x = F x = G r x` to hold, so `r = G r =: FIX G` and

`F := FIX G` where `FIX g := (r where r = g r) = g (FIX g)`

so that `FIX G = G (FIX G) = (λn.(1, if n = 0; else n × ((FIX G) (n−1))))`

Given a lambda term with first argument representing recursive call (e.g. `G` here), the *fixed-point* combinator `FIX` will return a self-replicating lambda expression representing the recursive function (here, `F`). The function does not need to be explicitly passed to itself at any point, for the self-replication is arranged in advance, when it is created, to be done each time it is called. Thus the original lambda expression `(FIX G)` is re-created inside itself, at call-point, achieving self-reference.

In fact, there are many possible definitions for this `FIX` operator, the simplest of them being:

```
Y := λg.(λx.g (x x)) (λx.g (x x))
```

In the lambda calculus, `Y g` is a fixed-point of `g`, as it expands to:

```
Y g
(λh.(λx.h (x x)) (λx.h (x x))) g
(λx.g (x x)) (λx.g (x x))
g ((λx.g (x x)) (λx.g (x x)))
g (Y g)
```

Now, to perform our recursive call to the factorial function, we would simply call `(Y G) n`, where *n* is the number we are calculating the factorial of. Given *n* = 4, for example, this gives:

```
(Y G) 4
G (Y G) 4
(λr.λn.(1, if n = 0; else n × (r (n−1)))) (Y G) 4
(λn.(1, if n = 0; else n × ((Y G) (n−1)))) 4
1, if 4 = 0; else 4 × ((Y G) (4−1))
4 × (G (Y G) (4−1))
4 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (4−1))
4 × (1, if 3 = 0; else 3 × ((Y G) (3−1)))
4 × (3 × (G (Y G) (3−1)))
4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (3−1)))
4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2−1))))
4 × (3 × (2 × (G (Y G) (2−1))))
4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (2−1))))
4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1−1)))))
4 × (3 × (2 × (1 × (G (Y G) (1−1)))))
4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (1−1)))))
4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0−1))))))
4 × (3 × (2 × (1 × (1))))
24
```

Every recursively defined function can be seen as a fixed point of some suitably defined function closing over the recursive call with an extra argument, and therefore, using **Y**, every recursively defined function can be expressed as a lambda expression. In particular, we can now cleanly define the subtraction, multiplication and comparison predicate of natural numbers recursively.

Standard terms

Certain terms have commonly accepted names:

```
I := λx.x
K := λx.λy.x
S := λx.λy.λz.x z (y z)
B := λx.λy.λz.x (y z)
C := λx.λy.λz.x z y
W := λx.λy.x y y
U := λx.λy.y (x x y)
ω := λx.x x
Ω := ω ω
Y := λg.(λx.g (x x)) (λx.g (x x))
```

Several of these have direct applications in the *elimination of lambda-abstraction* that turns lambda terms into combinator calculus terms.

Abstraction elimination

If *N* is a lambda-term without lambda-abstraction, but possibly containing named constants (combinators), then there exists a lambda-term *T*(*x*,*N*) which is equivalent to λ*x*.*N* but lacks lambda-abstraction (except as part of the named constants, if these are considered non-atomic). This can also be viewed as anonymising variables, as *T*(*x*,*N*) removes all occurrences of *x* from *N*, while still allowing argument values to be substituted into the positions where *N* contains an *x*. The conversion function *T* can be defined by:

T(*x*, *x*) := **I**
T(*x*, *N*) := **K** *N* if *x* is not free in *N*.
T(*x*, *M N*) := **S** *T*(*x*, *M*) *T*(*x*, *N*)

In either case, a term of the form $T(x,N)\,P$ can reduce by having the initial combinator **I**, **K**, or **S** grab the argument P, just like β-reduction of $(\lambda x\,.\,N)\,P$ would do. **I** returns that argument. **K** throws the argument away, just like $(\lambda x\,.\,N)$ would do if x has no free occurrence in N. **S** passes the argument on to both subterms of the application, and then applies the result of the first to the result of the second.

The combinators **B** and **C** are similar to **S**, but pass the argument on to only one subterm of an application (**B** to the "argument" subterm and **C** to the "function" subterm), thus saving a subsequent **K** if there is no occurrence of x in one subterm. In comparison to **B** and **C**, the **S** combinator actually conflates two functionalities: rearranging arguments, and duplicating an argument so that it may be used in two places. The **W** combinator does only the latter, yielding the B, C, K, W system as an alternative to SKI combinator calculus.

Typed lambda calculus

A **typed lambda calculus** is a typed formalism that uses the lambda-symbol (**λ**) to denote anonymous function abstraction. In this context, types are usually objects of a syntactic nature that are assigned to lambda terms; the exact nature of a type depends on the calculus considered (see kinds below). From a certain point of view, typed lambda calculi can be seen as refinements of the untyped lambda calculus but from another point of view, they can also be considered the more fundamental theory and *untyped lambda calculus* a special case with only one type.[22]

Typed lambda calculi are foundational programming languages and are the base of typed functional programming languages such as ML and Haskell and, more indirectly, typed imperative programming languages. Typed lambda calculi play an important role in the design of type systems for programming languages; here typability usually captures desirable properties of the program, e.g. the program will not cause a memory access violation.

Typed lambda calculi are closely related to mathematical logic and proof theory via the Curry–Howard isomorphism and they can be considered as the internal language of classes of categories, e.g. the simply typed lambda calculus is the language of Cartesian closed categories (CCCs).

Computable functions and lambda calculus

A function F: **N** → **N** of natural numbers is a computable function if and only if there exists a lambda expression f such that for every pair of x, y in **N**, $F(x)=y$ if and only if $f\,x =_\beta y$, where x and y are the Church numerals corresponding to x and y, respectively and $=_\beta$ meaning equivalence with beta reduction. This is one of the many ways to define computability; see the Church–Turing thesis for a discussion of other approaches and their equivalence.

Undecidability of equivalence

There is no algorithm that takes as input two lambda expressions and outputs TRUE or FALSE depending on whether or not the two expressions are equivalent. This was historically the first problem for which undecidability could be proven. As is common for a proof of undecidability, the proof shows that no computable function can decide the equivalence. Church's thesis is then invoked to show that no algorithm can do so.

Church's proof first reduces the problem to determining whether a given lambda expression has a *normal form*. A normal form is an equivalent expression that cannot be reduced any further under the rules imposed by the form. Then he assumes that this predicate is computable, and can hence be expressed in lambda calculus. Building on earlier work by Kleene and constructing a Gödel numbering for lambda expressions, he constructs a lambda expression e that closely follows the proof of Gödel's first incompleteness theorem. If e is applied to its own Gödel number, a contradiction results.

Lambda calculus and programming languages

As pointed out by Peter Landin's 1965 paper "A Correspondence between ALGOL 60 and Church's Lambda-notation"[23], sequential procedural programming languages can be understood in terms of the lambda calculus, which provides the basic mechanisms for procedural abstraction and procedure (subprogram) application.

Anonymous functions

For example, in Lisp the 'square' function can be expressed as a lambda expression as follows:

```
(lambda (x) (* x x))
```

The above example is an expression that evaluates to a first-class function. The symbol `lambda` creates an anonymous function, given a list of parameter names, `(x)` — just a single argument in this case, and an expression that is evaluated as the body of the function, `(* x x)`. Anonymous functions are sometimes called lambda expressions.

For example, Pascal and many other imperative languages have long supported passing subprograms as arguments to other subprograms through the mechanism of function pointers. However, function pointers are not a sufficient condition for functions to be first class datatypes, because a function is a first class datatype if and only if new instances of the function can be created at run-time. And this run-time creation of functions is supported in Smalltalk, JavaScript, and more recently in Scala, Eiffel ("agents"), C# ("delegates") and C++11, among others.

Reduction strategies

Whether a term is normalising or not, and how much work needs to be done in normalising it if it is, depends to a large extent on the reduction strategy used. The distinction between reduction strategies relates to the distinction in functional programming languages between eager evaluation and lazy evaluation.

Full beta reductions
: Any redex can be reduced at any time. This means essentially the lack of any particular reduction strategy—with regard to reducibility, "all bets are off".

Applicative order
: The rightmost, innermost redex is always reduced first. Intuitively this means a function's arguments are always reduced before the function itself. Applicative order always attempts to apply functions to normal forms, even when this is not possible.
Most programming languages (including Lisp, ML and imperative languages like C and Java) are described as "strict", meaning that functions applied to non-normalising arguments are non-normalising. This is done essentially using applicative order, call by value reduction (see below), but usually called "eager evaluation".

Normal order
: The leftmost, outermost redex is always reduced first. That is, whenever possible the arguments are substituted into the body of an abstraction before the arguments are reduced.

Call by name
: As normal order, but no reductions are performed inside abstractions. For example, $\lambda x.\,(\lambda x.x)x$ is in normal form according to this strategy, although it contains the redex $(\lambda x.x)x$.

Call by value
: Only the outermost redexes are reduced: a redex is reduced only when its right hand side has reduced to a value (variable or lambda abstraction).

Call by need
: As normal order, but function applications that would duplicate terms instead name the argument, which is then reduced only "when it is needed". Called in practical contexts "lazy evaluation". In implementations this "name" takes the form of a pointer, with the redex represented by a thunk.

Applicative order is not a normalising strategy. The usual counterexample is as follows: define **Ω** = ωω where **ω** = λ*x*.*xx*. This entire expression contains only one redex, namely the whole expression; its reduct is again **Ω**. Since this is the only available reduction, **Ω** has no normal form (under any evaluation strategy). Using applicative order, the expression **KIΩ** = (λ*x*.λ*y*.*x*) (λ*x*.*x*)**Ω** is reduced by first reducing **Ω** to normal form (since it is the rightmost redex), but since **Ω** has no normal form, applicative order fails to find a normal form for **KIΩ**.

In contrast, normal order is so called because it always finds a normalising reduction, if one exists. In the above example, **KIΩ** reduces under normal order to *I*, a normal form. A drawback is that redexes in the arguments may be copied, resulting in duplicated computation (for example, (λ*x*.*xx*) ((λ*x*.*x*)*y*) reduces to ((λ*x*.*x*)*y*) ((λ*x*.*x*)*y*) using this strategy; now there are two redexes, so full evaluation needs two more steps, but if the argument had been reduced first, there would now be none).

The positive tradeoff of using applicative order is that it does not cause unnecessary computation, if all arguments are used, because it never substitutes arguments containing redexes and hence never needs to copy them (which would duplicate work). In the above example, in applicative order (λ*x*.*xx*) ((λ*x*.*x*)*y*) reduces first to (λ*x*.*xx*)*y* and then to the normal order *yy*, taking two steps instead of three.

Most *purely* functional programming languages (notably Miranda and its descendents, including Haskell), and the proof languages of theorem provers, use *lazy evaluation*, which is essentially the same as call by need. This is like normal order reduction, but call by need manages to avoid the duplication of work inherent in normal order reduction using *sharing*. In the example given above, (λ*x*.*xx*) ((λ*x*.*x*)*y*) reduces to ((λ*x*.*x*)*y*) ((λ*x*.*x*)*y*), which has two redexes, but in call by need they are represented using the same object rather than copied, so when one is reduced the other is too.

A note about complexity

While the idea of beta reduction seems simple enough, it is not an atomic step, in that it must have a non-trivial cost when estimating computational complexity.[24] To be precise, one must somehow find the location of all of the occurrences of the bound variable *V* in the expression *E*, implying a time cost, or one must keep track of these locations in some way, implying a space cost. A naïve search for the locations of *V* in *E* is *O*(*n*) in the length *n* of *E*. This has led to the study of systems that use explicit substitution. Sinot's director strings[25] offer a way of tracking the locations of free variables in expressions.

Parallelism and concurrency

The Church–Rosser property of the lambda calculus means that evaluation (β-reduction) can be carried out in *any order*, even in parallel. This means that various nondeterministic evaluation strategies are relevant. However, the lambda calculus does not offer any explicit constructs for parallelism. One can add constructs such as Futures to the lambda calculus. Other process calculi have been developed for describing communication and concurrency

Optimal reduction

In Lévy's 1988 paper 'Sharing in the Evaluation of lambda Expressions", he defines a notion of optimal sharing, such that no work is *duplicated*. For example, performing a beta reduction in normal order on (λ*x*.*xx*) (II) reduces it to II (II). The argument II is duplicated by the application to the first lambda term. If the reduction was done in an applicative order first, we save work because work is not duplicated: (λ*x*.*xx*) (II) reduces to (λ*x*.*xx*) I. On the other hand, using applicative order can result in redundant reductions or even possibly never reduce to normal form. For example, performing a beta reduction in normal order on (λ*f*.f I) (λy.(λ*x*.*xx*) (y I)) yields (λy.(λ*x*.*xx*) (y I)) I, (λ*x*.*xx*) (II) which we know we can do without duplicating work. Doing the same but in applicative order yields (λ*f*.f I) (λy.y I (y I)), (λy.y I (y I)) I, I I (I I), and now work is duplicated.

Lévy shows the existence of lambda terms where there *does not exist* a sequence of reductions which reduces them without duplicating work. The below lambda term is such an example.

`((λg.(g(g(λx.x)))) (λh.((λf.(f(f(λz.z)))) (λw.(h(w(λy.y)))))))`

It is composed of three similar terms, `x=((λg. ... ) (λh.y))` and `y=((λf. ...) (λw.z) )`, and finally `z=λw.(h(w(λy.y)))`. There are only two possible beta reductions to be done here, on x and on y. Reducing the outer x term first results in the inner y term being duplicated, and each copy will have to be reduced, but reducing the inner y term first will duplicate its argument z, which will cause work to be duplicated when the values of h and w are made known. Incidentally, the above term reduces to the identity function `(λy.y)`, and is constructed by making wrappers which make the identity function available to the binders `g=λh...`, `f=λw...`, `h=λx.x` (at first), and `w=λz.z` (at first), all of which are applied to the innermost term `λy.y`.

The precise notion of duplicated work relies on noticing that after the first reduction of `I I` is done, the value of the other `I I` can be determined, because they have the same structure (and in fact they have exactly the same values), and result from a common ancestor. Such similar structures can each be assigned a label that can be tracked across reductions. If a name is assigned to the redex that produces all the resulting `II` terms, and then all duplicated occurrences of `II` can be tracked and reduced in one go. However, it is not obvious that a redex will produce the `II` term. Identifying the structures that are similar in different parts of a lambda term can involve a complex algorithm and can possibly have a complexity equal to the history of the reduction itself.

While Lévy defines the notion of optimal sharing, he does not provide an algorithm to do it. In Vincent van Oostrom, Kees-Jan van de Looij, and Marijn Zwitserlood's paper *Lambdascope: Another optimal implementation of the lambda-calculus*, they provide such an algorithm by transforming lambda terms into interaction nets, which are then reduced. Roughly speaking, the resulting reduction is optimal because every term that would have the same labels as per Lévy's paper would also be the same graph in the interaction net. In the paper, they mention that their prototype implementation of Lambdascope performs as well as the *optimised* version of the reference optimal higher order machine BOHM.

More details can be found in the short article About the efficient reduction of lambda terms.

Semantics

The fact that lambda calculus terms act as functions on other lambda calculus terms, and even on themselves, led to questions about the semantics of the lambda calculus. Could a sensible meaning be assigned to lambda calculus terms? The natural semantics was to find a set D isomorphic to the function space $D \to D$, of functions on itself. However, no nontrivial such D can exist, by cardinality constraints because the set of all functions from D to D has greater cardinality than D, unless D is a singleton set.

In the 1970s, Dana Scott showed that, if only continuous functions were considered, a set or domain D with the required property could be found, thus providing a model for the lambda calculus.

This work also formed the basis for the denotational semantics of programming languages.

See also

- Applicative computing systems – Treatment of objects in the style of the lambda calculus
- Binary lambda calculus – A version of lambda calculus with binary I/O, a binary encoding of terms, and a designated universal machine.
- Calculus of constructions – A typed lambda calculus with types as first-class values
- Cartesian closed category – A setting for lambda calculus in category theory
- Categorical abstract machine – A model of computation applicable to lambda calculus
- Combinatory logic – A notation for mathematical logic without variables
- Curry-Howard isomorphism – The formal correspondence between programs and proofs
- Deductive lambda calculus – The consideration of the problems associated with considering lambda calculus as a Deductive system.
- Domain theory – Study of certain posets giving denotational semantics for lambda calculus
- Evaluation strategy – Rules for the evaluation of expressions in programming languages
- Explicit substitution – The theory of substitution, as used in β-reduction
- Functional programming

- Harrop formula – A kind of constructive logical formula such that proofs are lambda terms
- Interaction nets
- Kappa calculus – A first-order analogue of lambda calculus
- Kleene-Rosser paradox – A demonstration that some form of lambda calculus is inconsistent
- Knights of the Lambda Calculus – A semi-fictional organization of LISP and Scheme hackers
- Krivine machine - A abstract machine to interpret call-by-name in lambda-calculus
- Lambda calculus definition - Formal definition of the lambda calculus.
- Lambda cube – A framework for some extensions of typed lambda calculus
- Lambda-mu calculus – An extension of the lambda calculus for treating classical logic
- Let expression – An expression closely related to a lambda abstraction.
- Minimalism (computing)
- Rewriting – Transformation of formulæ in formal systems
- SECD machine – A virtual machine designed for the lambda calculus
- Simply typed lambda calculus - Version(s) with a single type constructor
- SKI combinator calculus – A computational system based on the **S**, **K** and **I** combinators
- System F – A typed lambda calculus with type-variables
- Typed lambda calculus – Lambda calculus with typed variables (and functions)
- Universal Turing machine – A formal computing machine that is equivalent to lambda calculus
- Unlambda – An esoteric functional programming language based on combinatory logic

References

1. Turing, A. M. (December 1937). "Computability and λ-Definability" *The Journal of Symbolic Logic* **2** (4): 153–163. doi:10.2307/2268280 (https://doi.org/10.2307%2F2268280) JSTOR 2268280 (https://www.jstor.org/stable/2268280).
2. Coquand, Thierry, "Type Theory" (http://plato.stanford.edu/archives/sum2013/entries/type-theory/) *The Stanford Encyclopedia of Philosophy* (Summer 2013 Edition), Edward N. Zalta (ed.).
3. Moortgat, Michael (1988). *Categorial Investigations: Logical and Linguistic Aspects of the Lambek Calculus* (https://books.google.com/books?id=9CdFE9X_FCoC). Foris Publications. ISBN 9789067653879.
4. Bunt, Harry; Muskens, Reinhard, eds. (2008). *Computing Meaning* (https://books.google.com/books?id=nyFa5ngYThMC), Springer, ISBN 9781402059575
5. Mitchell, John C. (2003), *Concepts in Programming Languages* (https://books.google.com/books?id=7Uh8XGfJbEIC&pg=PA57), Cambridge University Press, p. 57, ISBN 9780521780988.
6. Pierce, Benjamin C. *Basic Category Theory for Computer Scientists* p. 53.
7. Church, A. (1932). "A set of postulates for the foundation of logic" *Annals of Mathematics* Series 2. **33** (2): 346–366. doi:10.2307/1968337 (https://doi.org/10.2307%2F1968337) JSTOR 1968337 (https://www.jstor.org/stable/1968337).
8. For a full history, see Cardone and Hindley's "History of Lambda-calculus and Combinatory Logic" (2006).
9. Kleene, S. C., Rosser, J. B. (July 1935). "The Inconsistency of Certain Formal Logics" *The Annals of Mathematics* **36** (3): 630. doi:10.2307/1968646 (https://doi.org/10.2307%2F1968646)
10. Church, Alonzo (December 1942). "Review of Haskell B. Curry, *The Inconsistency of Certain Formal Logics*". *The Journal of Symbolic Logic* **7** (4): 170–171. doi:10.2307/2268117 (https://doi.org/10.2307%2F2268117) JSTOR 2268117 (https://www.jstor.org/stable/2268117)
11. Church, A. (1936). "An unsolvable problem of elementary number theory" *American Journal of Mathematics* **58** (2): 345–363. doi:10.2307/2371045 (https://doi.org/10.2307%2F2371045) JSTOR 2371045 (https://www.jstor.org/stable/2371045).
12. Church, A. (1940). "A Formulation of the Simple Theory of Types". *Journal of Symbolic Logic* **5** (2): 56–68. doi:10.2307/2266170 (https://doi.org/10.2307%2F2266170) JSTOR 2266170 (https://www.jstor.org/stable/2266170)
13. Partee, B. B. H.; ter Meulen, A.; Wall, R. E. (1990). *Mathematical Methods in Linguistics* (https://books.google.com/books?id=qV7TUuaYcUIC&pg=PA317). Springer. Retrieved 29 Dec 2016.
14. Alama, Jesse "The Lambda Calculus" (http://plato.stanford.edu/entries/lambda-calculus/) *The Stanford Encyclopedia of Philosophy* (Summer 2013 Edition), Edward N. Zalta (ed.).

15. Barendregt, Hendrik Pieter (1984), *The Lambda Calculus: Its Syntax and Semantics* (https://web.archive.org/web/20040823174002/https://www.elsevier.com/wps/find/bookdescription.cws_home/501727/description), Studies in Logic and the Foundations of Mathematics, **103** (Revised ed.), North Holland, Amsterdam, ISBN 0-444-87508-5, archived from the original (http://www.elsevier.com/wps/find/bookdescription.cws_home/501727/description) on 2004-08-23 Corrections (ftp://ftp.cs.ru.nl/pub/CompMath.Found/ErrataLCalculus.pdf)
16. "Example for Rules of Associativity" (http://www.lambda-bound.com/book/lambdacalc/node27.html). Lambda-bound.com. Retrieved 2012-06-18.
17. Selinger, Peter (2008), *Lecture Notes on the Lambda Calculus* (http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf) (PDF), **0804** (class: cs.LO), Department of Mathematics and Statistics, University of Ottawa, p. 9, arXiv:0804.3434 (https://arxiv.org/abs/0804.3434), Bibcode:2008arXiv0804.3434S (http://adsabs.harvard.edu/abs/2008arXiv0804.3434S)
18. Barendregt, Henk; Barendsen, Erik (March 2000), *Introduction to Lambda Calculus* (ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf) (PDF)
19. de Queiroz, Ruy J. G. B. (1988). "A Proof-Theoretic Account of Programming and the Role of Reduction Rules". *Dialectica*. **42** (4): 265–282. doi:10.1111/j.1746-8361.1988.tb00919.x (https://doi.org/10.1111%2Fj.1746-8361.1988.tb00919.x).
20. Turbak, Franklyn; Gifford, David (2008), *Design concepts in programming languages*, MIT press, p. 251, ISBN 978-0-262-20175-9
21. Felleisen, Matthias; Flatt, Matthew (2006), *Programming Languages and Lambda Calculi* (https://web.archive.org/web/20090205113235/http://www.cs.utah.edu/plt/publications/pllc.pdf) (PDF), p. 26, archived from the original (http://www.cs.utah.edu/plt/publications/pllc.pdf) (PDF) on 2009-02-05; a note (accessed 2017) at the original location suggests that the authors consider the work originally referenced to have been superseded by a book.
22. Types and Programming Languages, p. 273, Benjamin C. Pierce
23. Landin, P. J. (1965). "A Correspondence between ALGOL 60 and Church's Lambda-notation" (http://portal.acm.org/citation.cfm?id=363749&coll=portal&dl=ACM). *Communications of the ACM* **8** (2): 89–101. doi:10.1145/363744.363749 (https://doi.org/10.1145%2F363744.363749)
24. Statman, R. (1979). "The typed λ-calculus is not elementary recursive" (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4567929). *Theoretical Computer Science* **9** (1): 73–81. doi:10.1016/0304-3975(79)90007-0 (https://doi.org/10.1016%2F0304-3975%2879%2990007-0)
25. Sinot, F.-R. (2005). "Director Strings Revisited: A Generic Approach to the Efficient Representation of Free Variables in Higher-order Rewriting" (http://www.lsv.ens-cachan.fr/~sinot/publis.php?onlykey=sinot-jlc05). *Journal of Logic and Computation*. **15** (2): 201–218. doi:10.1093/logcom/exi010 (https://doi.org/10.1093%2Flogcom%2Fexi010)

Further reading

- Abelson, Harold & Gerald Jay Sussman. Structure and Interpretation of Computer Programs. The MIT Press. ISBN 0-262-51087-1.
- Hendrik Pieter Barendregt. *Introduction to Lambda Calculus*
- Henk Barendregt. The Impact of the Lambda Calculus in Logic and Computer Science. The Bulletin of Symbolic Logic, Volume 3, Number 2, June 1997.
- Barendregt, Hendrik Pieter. *The Type Free Lambda Calculus*. pp1091–1132 of *Handbook of Mathematical Logic*, North-Holland (1977) ISBN 0-7204-2285-X
- Cardone and Hindley, 2006. History of Lambda-calculus and Combinatory Logic. In Gabbay and Woods (eds.), *Handbook of the History of Logic*, vol. 5. Elsevier.
- Church, Alonzo, *An unsolvable problem of elementary number theory*, American Journal of Mathematics, 58 (1936), pp. 345–363. This paper contains the proof that the equivalence of lambda expressions is in general not decidable.
- Alonzo Church, *The Calculi of Lambda-Conversion* (ISBN 978-0-691-08394-0)[1]
- Kleene, Stephen, *A theory of positive integers in formal logic*, American Journal of Mathematics, 57 (1935), pp. 153–173 and 219–244. Contains the lambda calculus definitions of several familiar functions.
- Landin, Peter, *A Correspondence Between ALGOL 60 and Church's Lambda-Notation*, Communications of the ACM, vol. 8, no. 2 (1965), pages 89–101. Available from the ACM site. A classic paper highlighting the importance of lambda calculus as a basis for programming languages.
- Larson, Jim, *An Introduction to Lambda Calculus and Scheme*. A gentle introduction for programmers.
- Schalk, A. and Simmons, H. (2005) *An introduction to λ-calculi and arithmetic with a decent selection of exercises Notes for a course in the Mathematical Logic MSc at Manchester University*

- de Queiroz, Ruy J.G.B. (2008) "On Reduction Rules, Meaning-as-Use and Proof-Theoretic Semantics" *Studia Logica*, **90**(2):211-247. doi:10.1007/s11225-008-9150-5 A paper giving a formal underpinning to the idea of 'meaning-is-use' which, even if based on proofs, it is different from proof-theoretic semantics as in the Dummett–Prawitz tradition since it takes reduction as the rules giving meaning.
- Hankin, Chris, *An Introduction to Lambda Calculi for Computer Scientists*, ISBN 0954300653

Monographs/textbooks for graduate students:

- Morten Heine Sørensen, Paweł Urzyczyn, *Lectures on the Curry-Howard isomorphism*, Elsevier, 2006, ISBN 0-444-52077-5 is a recent monograph that covers the main topics of lambda calculus from the type-free variety, to most typed lambda calculi, including more recent developments like pure type systems and the lambda cube. It does not cover subtyping extensions.
- Pierce, Benjamin (2002), *Types and Programming Languages*, MIT Press, ISBN 0-262-16209-1 covers lambda calculi from a practical type system perspective; some topics like dependent types are only mentioned, but subtyping is an important topic.

Some parts of this article are based on material from FOLDOC, used with permission.

External links

- Graham Hutton, Lambda Calculus, a short (12 minutes) Computerphile video on the Lambda Calculus
- Hazewinkel, Michiel, ed. (2001) [1994], "Lambda-calculus", *Encyclopedia of Mathematics*, Springer Science+Business Media B.V / Kluwer Academic Publishers, ISBN 978-1-55608-010-4
- Achim Jung, *A Short Introduction to the Lambda Calculus* (PDF)
- Dana Scott, *A timeline of lambda calculus* (PDF)
- David C. Keenan, *To Dissect a Mockingbird: A Graphical Notation for the Lambda Calculus with Animated Reduction*
- Raúl Rojas, *A Tutorial Introduction to the Lambda Calculus* (PDF)
- Peter Selinger, *Lecture Notes on the Lambda Calculus* (PDF)
- L. Allison, *Some executable λ-calculus examples*
- Georg P. Loczewski, *The Lambda Calculus and A++*
- Bret Victor, *Alligator Eggs: A Puzzle Game Based on Lambda Calculus*
- *Lambda Calculus* on Safalra's Website
- *"Lambda Calculus"*. PlanetMath.
- LCI Lambda Interpreter, a simple yet powerful pure calculus interpreter
- Lambda Calculus links on Lambda-the-Ultimate
- Mike Thyer, Lambda Animator, a graphical Java applet demonstrating alternative reduction strategies.
- Implementing the Lambda calculus using C++ Templates
- Marius Buliga, *Graphic lambda calculus*
- *Lambda Calculus as a Workflow Model* by Peter Kelly, Paul Coddington, and Andrew Wendelborn; mentions graph reduction as a common means of evaluating lambda expressions and discusses the applicability of lambda calculus for distributed computing (due to the Church–Rosser property, which enables parallel graph reduction for lambda expressions).
- Shane Steinert-Threlkeld, "Lambda Calculi", *Internet Encyclopedia of Philosophy*
- Anton Salikhmetov, *Macro Lambda Calculus*

1. Frink Jr., Orrin (1944). "Review: *The Calculi of Lambda-Conversion* by Alonzo Church" (http://www.ams.org/bull/1944-50-03/S0002-9904-1944-08090-7/S0002-9904-1944-08090-7.pdf) (PDF). *Bull. Amer. Math. Soc.* **50** (3): 169–172. doi:10.1090/s0002-9904-1944-08090-7 (https://doi.org/10.1090%2Fs0002-9904-1944-08090-7)