lam.1 Introduction

cmp:lam:int:sec

The lambda calculus was originally designed by Alonzo Church in the early 1930s as a basis for constructive logic, and *not* as a model of the computable functions. But soon after the Turing computable functions, the recursive functions, and the general recursive functions were shown to be equivalent, lambda computability was added to the list. The fact that this initially came as a small surprise makes the characterization all the more interesting.

Lambda notation is a convenient way of referring to a function directly by a symbolic expression which defines it, instead of defining a name for it. Instead of saying "let f be the function defined by $f(x) = x + 3$," one can say, "let f be the function $\lambda x.\,(x + 3)$." In other words, $\lambda x.\,(x + 3)$ is just a *name* for the function that adds three to its argument. In this expression, x is a dummy variable, or a placeholder: the same function can just as well be denoted by $\lambda y.\,(y+3)$. The notation works even with other parameters around. For example, suppose $g(x, y)$ is a function of two variables, and k is a natural number. Then $\lambda x.\,g(x, k)$ is the function which maps any x to $g(x, k)$.

This way of defining a function from a symbolic expression is known as *lambda abstraction*. The flip side of lambda abstraction is *application*: assuming one has a function f (say, defined on the natural numbers), one can *apply* it to any value, like 2. In conventional notation, of course, we write $f(2)$ for the result.

What happens when you combine lambda abstraction with application? Then the resulting expression can be simplified, by "plugging" the applicand in for the abstracted variable. For example,

$$(\lambda x.\,(x + 3))(2)$$

can be simplified to $2 + 3$.

Up to this point, we have done nothing but introduce new notations for conventional notions. The lambda calculus, however, represents a more radical departure from the set-theoretic viewpoint. In this framework:

1. Everything denotes a function.
2. Functions can be defined using lambda abstraction.
3. Anything can be applied to anything else.

For example, if F is a term in the lambda calculus, $F(F)$ is always assumed to be meaningful. This liberal framework is known as the *untyped* lambda calculus, where "untyped" means "no restriction on what can be applied to what."

digression

There is also a *typed* lambda calculus, which is an important variation on the untyped version. Although in many ways the typed lambda calculus is similar to the untyped one, it is much easier to reconcile with a classical set-theoretic framework, and has some very different properties.

Research on the lambda calculus has proved to be central in theoretical computer science, and in the design of programming languages. LISP, designed by John McCarthy in the 1950s, is an early example of a language that was influenced by these ideas.

Photo Credits

Bibliography