Theory of Formal Languages (Lambda Calculus)

Lecture 14: Feb. 23, 2019

Prof. K.R. Chowdhary *: Professor of CS*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

14.1 Introduction

Since the meaning of a computer program can be considered as a function from input values to output values, the functions play a prominent role in describing the semantics of a programming language, Alonzo Church developed the lambda calculus in the 1930s as a theory of functions that provides rules for manipulating functions in a purely *syntactic* manner. The λ-calculus is the simplest programming language. Although the lambda calculus arose as a branch of mathematical logic to provide a foundation for mathematics, it has led to considerable ramifications in the theory of programming languages. Beyond the influence of the lambda calculus in the area of computation theory, it has contributed important results to the formal semantics of programming languages in the following ways:

- Although the lambda calculus has the power to represent all computable functions, its uncomplicated syntax and semantics provide an excellent vehicle for studying the meaning of programming language concepts.
- All functional programming languages can be viewed as syntactic variations of the lambda calculus, so that both their semantics and implementation can be analyzed in the context of the lambda calculus.
- Denotational semantics, one of the foremost methods of formal specification of languages, grew out of research in the lambda calculus and expresses its definitions using the higher-order functions of the lambda calculus.

Some of the features of λ-calculus are as follows:

- The λ-calculus can be used to encode programs AND data, such as Booleans and natural numbers,
- It is the simplest possible programming language, that is Turing complete,
- 'Pure LISP' is equivalent Lambda Calculus, and
- 'LISP' is Lambda calculus, plus some additional features such as data types, input/output, etc.

The Popular question in 1930's was: "What does it mean for a function f to be computable?" The Intuitive computability was defined as : A pencilandpaper method can be used to allow a trained person to calculate $f(n)$ for any given n? This was further defined by Turing, *Gödel*, and Church as follows.

- Turing: A function is computable if and only if it can be computed by the Turing machine.
- *Gödel*: A function is computable if and only if it is general recursive.
- Church: A function is computable if it can be written as a lambda term.

It has been proven that all three models are equivalent. However, the question – "Are they equivalent to 'intuitive computability'?" Cannot be answered.

14.2 Functional Programming languages

A mathematical function is a mapping of domain set to range set. A function definition is the description of this mapping either explicitly by enumeration or implicitly by an expression. The definition of a function is specified by a function name followed by a list of parameters in parenthesis, followed by the expression describing the mapping, e.g., $cube(X) \equiv X * X * X$, where X is a real number. Alonzo Church introduced the notation of nameless functions using the Lambda notation. A lambda expression specifies the parameters and the mapping of a function using the λ operator, e.g., $\lambda(X)X * X * X$. It is the function itself, so the notation of applying the example nameless function to a certain argument is, for example, $(\lambda(X)X * X * X)(4)$.

Programming in a functional language consists of building function definitions and using the computer to evaluate expressions, i.e. function application with concrete arguments. The major programming task is then to construct a function for a specific problem by combining previously defined functions according to mathematical principles. The main task of the computer is to evaluate function calls and to print the resulting function values. This way the computer is used like an ordinary pocket computer, of course at a much more flexible and powerful level. A characteristic feature of functional programming is that if an expression possesses a well-defined value, then the order in which the computer performs the evaluation does not affect the result of the evaluation. Thus, the result of the evaluation of an expression is just its value. This means that in a pure functional language no side-effects exist. Side-effects are connected to variables that model memory locations. Thus, in a pure functional programming language no variables exists in the sense of imperative languages. The major control flow methods are recursion and conditional expressions. This is quite different from imperative languages, in which the basic means for control are sequencing and iteration. Functional programming also supports the specification of higher-order functions. A higherorder function is a function definition which allows functions as arguments or returns a function as its value.

All these aspects together, but especially the latter are major sources of the benefits of functional programming style in contrast to imperative programming style, viz. that functional programming provides a high-level degree of modularity. When defining a problem by dividing it into a set of sub-problems, a major issue concerns the ways in which one can glue the (sub-) solutions together. Therefore, to increase ones ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language – a major strength of functional programming.

A functional program consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules. Reduction consists of replacing a part P of E by another expression P' according to the given rewrite rules. In schematic notation,

$$E[P] \rightarrow E[P'],$$

provided that $P \rightarrow P'$ is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called normal form E of the expression E, and consists of the output of the given functional program. An example is,

$$\begin{aligned}(7+4)*(8+5*3) &\rightarrow 11*(8+5*3)\\ &\rightarrow 11*(8+15)\\ &\rightarrow 11*23\\ &\rightarrow 253.\end{aligned}$$

In this example the reduction rules consist of the 'tables'of addition and of multiplication on the numerals. Also symbolic computations can be done by reduction. For example,

first of (sort (append ('dog', 'rabbit') (sort (('mouse', 'cat')))))
$\rightarrow$ first of (sort (append ('dog', 'rabbit') ('cat', 'mouse')))
$\rightarrow$ first of (sort ('dog', 'rabbit', 'cat', 'mouse'))
$\rightarrow$ first of ('cat', 'dog', 'mouse', 'rabbit')
$\rightarrow$ 'cat'.

The necessary rewrite rules for *append* and *sort* can be programmed easily in a few lines. Functions like append given by some rewrite rules are called *combinators*.

Reduction systems usually satisfy the *Church-Rosser* property, which states that the normal form obtained is independent of the order of evaluation of subterms. Indeed, the first example may be reduced in another style as follows.

$$\begin{aligned}(7+4)*(8+5*3) &\rightarrow (7+4)*(8+15)\\ &\rightarrow (7+4)*23\\ &\rightarrow 11*23\\ &\rightarrow 253.\end{aligned}$$

14.3 Lambda Calculus and Computability

In the 1930's, several people were interested in the question: what does it mean for a function $f : \mathbb{N} \rightarrow \mathbb{N}$ to be computable? An informal definition of computability is that there should be a *pencil-and-paper* method allowing a trained person to calculate $f(n)$, for any given n. The concept of a pencil-and-paper method is not so easy to formalize. Three different researchers attempted to do so, resulting in the following definitions of computability:

1. Turing defined an idealized computer we now call a *Turing machine*, and postulated that a function is computable (in the intuitive sense) if and only if it can be computed by such a machine.

2. *Gödel* defined the class of *general recursive functions* as the smallest set of functions containing all the constant functions, the successor function, and closed under certain operations (such as compositions and recursion). He postulated that a function is computable (in the intuitive sense) if and only if it is general recursive.

3. Church defined an idealized programming language called the *lambda calculus*, and postulated that a function is computable (in the intuitive sense) *if and only if* it can be written as a lambda term.

It was proved by Church, Kleene, and Turing that all three computational models were equivalent to each other, i.e., each model defines the same class of computable functions. Whether or not they are equivalent to the "intuitive" notion of computability is a question that cannot be answered, because there is no formal definition of "intuitive computability". The assertion that they are in fact equivalent to intuitive computability is known as the Church-Turing thesis.

14.4 Basic Concepts of λ-Calculus

The lambda calculus is a theory of functions as formulas. It is a system for manipulating functions as expressions. Let us begin by looking at another well-known language of expressions, namely arithmetic. Arithmetic expressions are made up from variables $(a, b, ..., x, y, z, x_0, x_1, \ldots)$, numbers $(1, 2, 3, \ldots)$, and operators $\{+, , \times\}$ etc. An expression such as $x + y$ stands for the result of an addition (as opposed to an instruction to add or the statement that something is being added).

The great advantage of this language is that expressions can be nested without any need to mention the intermediate results explicitly. So for instance, we write

$$A = (x + y) \times z^2,$$

and not the following: let $w = x + y$, then let $u = z^2$, then let $A = w \times u$. The latter notation would be tiring and cumbersome to manipulate. The *lambda calculus* extends the idea of an expression language to include functions, where we normally write,

$$A = (\lambda x.x^2)(5).$$

The expression $\lambda x.x^2$ stands for the function that maps x to x^2 (as opposed to the statement that x is being mapped to x^2). As in arithmetic, we use parentheses to group terms. It is understood that the variable x is a local variable in the term $\lambda x.x^2$. Thus, it does not make any difference if we write $\lambda y.y^2$ instead. A local variable is also called a *bound* variable.

The set of lambda terms Λ is built up from variables using *abstraction*

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

and *application*

$$(M \;\; N),$$

where x is any variable and M, N are lambda terms. $(\lambda x.M)$ is the function that maps x to M, while $(M \;\; N)$ is application of function M to argument N. We sometimes omit parenthesis, understanding abstraction to associate to right, and application to associate to left. For example, $\lambda x.\lambda y.x \; y \; x$ denotes $(\lambda x.(\lambda y.((x \; y)x)))$. We also join the consecutive abstractions as in $\lambda x \; y.x \; y \; x$.

We consider two terms identical if they only differ in names of bound variables, and denote this with $\equiv$, e.g., $\lambda x.y\ x \equiv \lambda z.y\ \ z$. The essence of *lambda*-calculus is embodied in reduction (called β-conversion) rule, which equates,

$$(\lambda x.M)N = M[x := N],$$

where $M[x := N]$ denotes the result of substituting N for all free occurrences of x in M. For example,

$$(\lambda xy.y\ x)y \equiv (\lambda x.(\lambda z.z\ x))y \equiv (\lambda x\ z.z\ x)y = \lambda z.z\ y.$$

A term with no possible β-reduction, i.e., no subterm of the form $(\lambda.x\ M)N$, is called *normal form*. The terms may be viewed as denoting computations of which β-reduction form the steps, and which may halt with a normal form as the end of the result.

The Church's lambda notation allows the definition of an *anonymous function*, that is, a function without a name: For example, $\lambda n.n^3$ defines the function that maps each n in the domain to n^3.

We say that the expression represented by $\lambda n.n^3$ is the value bound to the identifier "cube". The number and order of the parameters to the function are specified between the λ symbol and an expression. For instance, the expression $n^2 + m$ is ambiguous as the definition of a function rule:

$$(3,5) \mapsto 3^2 + 5 = 14$$

or

$$(3,5) \mapsto 5^2 + 3 = 28.$$

Lambda notation resolves the ambiguity by specifying the order of the parameters:

$$\lambda n.\lambda m.n^2 + m$$

or

$$\lambda m.\lambda n.n^2 + m.$$

14.4.1 Expressions in the λ-Calculus

The λ-calculus is a notation for functions. It is extremely economical but at first sight perhaps somewhat cryptic, which stems from its origins in mathematical logic. Expressions in the λ-calculus are written in strict prefix form, that is, there are no infix or postfix operators (such as $+, , (^2_)$, etc.). Furthermore, function and argument are simply written next to each other, without brackets around the argument. So where the mathematician and the computer programmer would write "*sin*(x)", in the λ-calculus we simply write "*sin x*". If a function takes more than one argument, then these are simply lined up after the function. Thus "$x+3$" becomes "$+\ x\ 3$", and "x^2" becomes "$*\ x\ x$". Brackets are employed only to enforce a special grouping. For example, where we would normally write "*sin*$(x)+4$", the λ-calculus formulation is "$+\ (sin\ x)\ 4$".

14.4.2 Application and abstraction

The first basic operation of the λ-calculus is application. The expression $F.A$ or FA denotes the data F considered as algorithm applied to the data A considered as input. This can be viewed in two ways: either as the process of computation FA or as the output of this process. The first view is captured by the notion of conversion and even better of reduction; the second by the notion of models (semantics).

The theory is *type-free* - it is allowed to consider expressions like F,F, that is F applied to itself. This will be useful to simulate recursion. The other basic operation is abstraction. If $M \equiv M[x]$ is an expression containing ('depending on') x, then $\lambda x.M[x]$ denotes the function $x \to M[x]$.

Application and abstraction work together in the following intuitive formula.

$$\begin{aligned}(\lambda x.2x+1)3 &\to 23+1\\ &\to 7.\end{aligned}$$

That is, $(\lambda x.2x+1)3$ denotes the function $x \to 2x+1$ when applied to the argument 3 gives a result $23+1$ which is 7. In general we have

$$(\lambda x.M\ [x])N = M[N]. \tag{14.1}$$

This last equation is preferably written as

$$(\lambda x.M)N = M[x := N] \tag{14.2}$$

where $[x := N]$ denotes substitution of N for x. It is remarkable that although above is the only essential axiom of the λ-calculus, the resulting theory is rather involved.

14.4.3 Functions in the λ-calculus

If an expression contains a variable x, then one can form the function which is obtained by considering the relationship between concrete values for x and the resulting value of the expression. In mathematics, function formation is written as an equation, e,g, $f(x) = 3x$. The λ-calculus dispenses with the need to give a name to the function (like in $f(x) = 3x$) and it easily scales up to more complicated function definitions. For example, we would re-write the expression "3x" into " 3 x" and then turn it into a function by preceding it with λx.". We get: "$\lambda x.\ *\ 3\ x$". The symbol λ has a role similar to the keyword "function" in some programming languages. It alerts the reader that the variable which follows is not part of an expression but the formal parameter of the function declaration. The dot (.) after the formal parameter introduces the function body.

A function which has been written in λ-notation can itself be used in an expression. For example, the application of the function from above to the value 4 is written as $(\lambda x.*\ 3\ x)\ 4$. Note that, application is simply juxtaposition; but why the brackets are there around the function? They are to make clear where the definition of the function ends. If the above is written as $\lambda x.*\ 3\ x\ 4$ then the value 4 becomes the part of the function body and we would get the function which assigns to x the value $3\ *\ x\ *\ 4$, which is not having any meaning. So, the bracket is used to delineating parts of λ-term, but these brackets have no meaning in it self.

We can introduce abbreviations for λ-terms, which are used in the same way as we do in mathematics, employing equality symbol. So, if we want to abbreviate the our function term to F:

$$F \overset{\text{def}}{=} \lambda x.\ *\ 3\ x \tag{14.3}$$

then, we can write F 4 instead of $(\lambda x. *\ 3\ x)\ 4$.

Consider that body of a function consist another function, the λ-calculus notation is as follows:

$$N \overset{\text{def}}{=} \lambda y.(\lambda x.\ y\ x) \tag{14.4}$$

if we apply this function to the value 3 then we get back to the old expression $\lambda x. *\ 3\ x$. In other words, N is a function, which when applied to a number, returns another function (i.e., N 3 behaves like F). However, we could also consider it as a function of two arguments, such that we get a number back if we supply N with two numerical arguments (N 3 4 should evaluate to 12). Both the views are correct. If we stress the first interpretation we may write the term with brackets as above, and if we want to see it as a function of two arguments then we can leave out the brackets, as,

$$\lambda y.\lambda x.y\ x$$

or, as follows, where we skip the 2nd lambda,

$$\lambda y\ x. *\ y\ x.$$

In the application of N to arguments 3 and 4 we can use brackets to stress that 3 is to be used first: $(N\ \ 3)\ 4$ or we can suggest simultaneous application: N 3 4. What ever is out intuition, the final result will be same in both the cases, i.e., 12.

14.4.4 Reduction

The λ-terms have only one rule of computation – called *reduction* (or β-reduction), and it concerns the replacement of a formal parameter by an actual parameter. It can only occur if a functional term has been applied to some other term. Examples are:

$$(\lambda x. *\ 3\ x)\ 4 \rightarrow_{\beta}\ \ *\ 3\ 4$$

$$(\lambda y.y\ 5)(\lambda x. *\ 3\ x) \rightarrow_{\beta} (\lambda x. *\ 3\ x)\ 5 \rightarrow_{\beta} *\ 3\ 5$$

We see that reduction is nothing other than the textual replacement of a formal parameter in the body of a function by the actual parameter supplied.

One would expect a term after a number of reductions to reach a form where no further reductions are possible. Surprisingly, this is not always the case. The following is the smallest counter example:

$$\Omega \overset{\text{def}}{=} (\lambda x.x.x)(\lambda x.x.x) \tag{14.5}$$

The term Ω always reduces to itself. If a sequence of reductions has come to an end where no further reductions are possible, we say that the term has been reduced to normal form. As Ω illustrates, not every term has a normal form.

14.4.5 Higher-order functions

The λ-calculus is a purely syntactic device; it does not make any distinctions between simple entities, such as numbers and more complicated ones, such as functions of functions. Whatever can be described as a λ-term is available for manipulation by other λ-terms.

Let us look at an example. A term for squaring integers is given by

$$Q \overset{\text{def}}{=} \lambda x. * \; x \; x \tag{14.6}$$

If we want to compute x^8 then this can be achieved by squaring x three times: $x^8 = ((x^2)^2)^2$. In λ-calculus notation, we would write for the "power-8"-function:

$$P_8 \overset{\text{def}}{=} \lambda x.Q(Q(Q\,x)) \tag{14.7}$$

We see that taking a number to power 8 amounts to applying the squaring function Q three times. It is now a simple step to write out a λ-term which applies any function three times:

$$T \overset{\text{def}}{=} \lambda f.(\lambda x.f(f(f\;x))) \tag{14.8}$$

Make a note of unnecessary brackets around the inner function; note that T takes an argument a function f and returns another function with argument x. The term P_8 can now be written as $T\;Q$, and 5^8 comes out as $T\;Q\;5$.

There is nothing to stop us from applying the tripling operator T. What we get is, an operator which will triple any function we pass to it three times, so it is in fact a 27-fold operator, that is $T\;T\;f\;x$ will compute the result of applying f 27 times to x. The operator, like T are called *higher order* because they operate on functions rather than numbers.

14.4.6 Iteration and recursion

First, a simplifying assumption: we will only deal with the set of natural numbers. That is, all integers not less than zero, or to put it in better way, a natural number is zero, or one plus the natural number. Then, what is number? A number is a count of stuff. We consider,

$$\begin{array}{rll} N = & 0 & \text{0 function applications} \\ | & 1+N & \text{1 more function application} \end{array}$$

What function should we use? It does not really matter. Consider the following:

$$\begin{aligned}
0 &\equiv \lambda f.\lambda x.x \\
1 &\equiv \lambda f.\lambda x.f\ x \\
2 &\equiv \lambda f.\lambda x.f(f\ x) \\
N &\equiv \lambda f.\lambda x.f^N\ x
\end{aligned}$$

The above numeric representation is known as *Church numeral*.

As we have seen with the terms T and $T\ T$, a short combination of λ-terms can express repeated application of a function. How can we generalize this to get the behavior of a for-loop, where the number of repetitions is controlled by a counter? This requires a wholly new idea which we will now develop step by step.

First of all, we have to use a constant which allows us to distinguish between 0 and positive numbers. Let us call this constant “zero?”. Its behaviour is like an if-then-else clause depending on the value of a number:

$$\begin{aligned}
&zero?\ 0\ x\ y\ \rightarrow\ x \\
&zero?\ n\ x\ y\ \rightarrow\ y \quad (n \neq 0)
\end{aligned}$$

We also assume constants ”pred“ and ”succ“ for predecessor and successor function on natural numbers.

Let us now construct a term I (for ”Iteration“) which takes as arguments a number n, a function f, and a value x, and computes the n-fold application of f to x:

$$I\ n\ f\ x = f(f(f...(f\ x)...))$$

If $n = 0$ then $I\ 0\ f\ x$ should simply return x, without applying f at all. Here is a first attempt at defining I:

$$I = \lambda n\ f\ x.zero?\ n\ x\ (I\ (pred\ n)\ f\ (f\ x))$$

Here is the rationale: If $n = 0$ then zero? $n\ x\ M$ will evaluate to x, no matter what M is. If $n > 0$ then we iterate f $(n-1)$-times on the argument $(f\ x)$; if successful, this will return f applied to x n-times.

14.5 Syntax and semantics of Lambda Calculus

Like other programming languages the lambda calculus has syntax and semantics, though each very simple and unambiguous.

14.5.1 Syntax of lambda calculus

The lambda calculus derives its usefulness from having a sparse syntax and a simple semantics, and yet it retains sufficient power to represent all computable functions. Lambda expressions are in four varieties:

1. *Variables*, which are usually taken to be any lowercase letters.
2. *Predefined constants*, which act as values and operations are allowed in an impure or applied lambda calculus .
3. *Function applications* (combinations).
4. *Lambda abstractions* (function definitions).

A lambda calculus syntax is represented as a BNF specification:

$$\begin{aligned} < expression > ::= & \langle x \rangle & \text{(variables)} \\ & \mid \langle \lambda x.e \rangle & \text{(function)} \\ & \mid \langle e.e \rangle & \text{(function application)} \end{aligned} \tag{14.9}$$

where *variables* are lowercase identifiers, *constants* are predefined objects, $(\langle e \rangle.\langle e \rangle)$ is combinations, and $(\lambda \langle x \rangle.\langle e \rangle)$ is abstractions.

We allow identifiers of more than one letter to stand as variables and constants. The pure lambda calculus has no predefined constants, but it still allows the definition of all of the common constants and functions of arithmetic and list manipulation. In our explanations, we will allow predefined constants, including numerals (for example, 35), *add* (for addition), *mul* (for multiplication), *succ* (the successor function), and *sqr* (the squaring function).

In an abstraction, the variable named is referred to as the *bound* variable and the associated lambda expression is the *body* of the abstraction. With a function application $(E_1 E_2)$, it is assumed that E_1 evaluates to a predefined function (a constant) or an abstraction, say $(\lambda x.E_3)$, in which case the result of the application will be the evaluation of E_3 after every "free" occurrence of x in E_3 has been replaced by E_2. In a combination $(E_1 E_2)$, the function or operator E_1 is called the *rator* and its argument or operand E_2 is called the *rand*.

The function formation and functions are every thing in the lambda calculus. They can be mixed freely and used as often as desired, which is another way of saying that λ-terms are constructed according to the grammar,

$$M ::= c \mid x \mid M \;\; M \mid \lambda x.M$$

Here, the placeholder c represent any **constant** we might wish to use in a λ-term, such as number $1, 2, 3, ...$ or arithmetic operators $+$, $*$, etc. A term without constants is called **pure**. Similarly, the letter x represents any infinitely many possible variables. The grammar is ambiguous; the term $\lambda x.x\, y$ could be parsed as shown in figure 14.1.

In the above figure "app“ indicates the clause "$M \;\; M$“ in the derivation. With auxiliary brackets, the two possible interpretations can be indicated by writing $\lambda x.(x\, y)$ and $(\lambda .x)\, y$, respectively. according

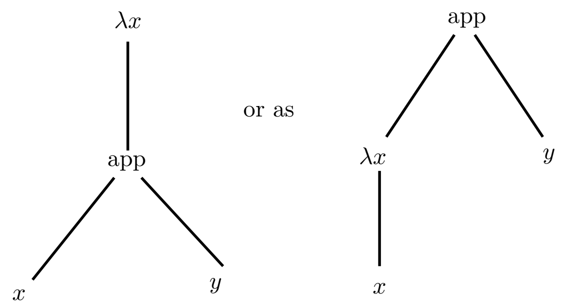

Figure 14.1: Parsing of λ-expression.

to the convention above, only the first interpretation should be possible. Using the additional non-terminals and productions the conventional interpretation can be enforced as follows:

$$\begin{aligned}
< term > &::= < atom > | < app > | < fun > \\
< atom > &::= < head - atom > | \ (< app >) \\
< head - atom > &::= x \mid c \mid (< fun >) \\
< app > &::= < head - atom >< atom > | < app >< atom > \\
< fun > &::= \lambda x. < term >
\end{aligned}$$

Note that, these are the rules of grammar of λ-calculus, that are implemented by compiler.

14.5.2 Semantics of Lambda calculus

Consider evaluating of function application: $(\lambda x.e_1)e_2$. Following are the semantics expressed by this expression:

1. Replace every x in e_1 with e_2
2. Evaluate the resulting terms
3. Return the result of computation

The "β-reduction" can be formally expressed as,

$$(\lambda x.e_1)e_2 \rightarrow_\beta e_1[e_2/x].$$

The λ extends as far right to as possible, for example, in following:

$$\lambda x.\lambda y.xy$$

is equal to,

$$\lambda x.(\lambda y.(xy)).$$

In the above, the function application is always *left-associative*.

Following is procedure for function with many arguments: We cannot write function with many arguments. For example, for two arguments: $\lambda(x,y).e$. The solution is to take one argument at a time, as follows:

$$\lambda x.\lambda y.e$$

A function that takes x and returns another function that takes y and returns e is:

$$(\lambda x.\lambda y.e)ab \rightarrow (\lambda e.e[a/x]b \rightarrow e[a/x][b/y].$$

This is called *currying*.

14.6 Complex Functions

In this category comes the composition of functions, i.e., a function calling or having another function as argument, and functions with multiple arguments.

Composition of Functions. One advantage of the lambda notation is that it allows us to easily talk about higher-order functions, i.e., functions whose inputs and/or outputs are themselves functions. An example is the operation $f \mapsto f \circ f$ in mathematics, which takes a function f and maps it to $f \circ f$, the composition of f with itself. In the lambda calculus, $f \circ f$ is written as

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

and the operation that maps f to $f \circ f$ is written as

$$\lambda f.\lambda x.f(f(x)).$$

The evaluation of higher-order functions can get somewhat complex. As an example, consider the following expression:

$$((\lambda f.\lambda x.f(f(x)))(\lambda y.y^2))(5)$$

Convince yourself that above evaluates to 625.

Functions with more than one arguments. Functions of several arguments can be obtained by iteration of application. Intuitively, if $f(x,y)$ depends on two arguments, one can define

$$F_x = \lambda y.f(x,y),$$

$$F = \lambda x.F_x.$$

Then

$$(F_x)y = F_x y = f(x,y).$$

This last equation shows that it is convenient to use association to the left for iterated application: $FM_1 \dots M_n$ denotes

$$(\dots((FM_1)M_2)\dots M_n).$$

The equation above becomes

$$Fxy = f(x,y).$$

Dually, iterated abstraction uses association to the right: $\lambda x_1 \dots x_n.f\ (x_1,\dots,x_n)$ denotes,

$$\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.f(x_1,\dots,x_n)))).$$

14.7 Examples

Lambda expressions will be illustrated with several examples in which we use prefix notation as in Lisp for predefined binary operations and so avoid the issue of precedence among operations.

- The lambda expression $\lambda x.x$ denotes the identity function in the sense that $((\lambda x.x)E) = E$ for any lambda expression E. The lambda expression $(\lambda x.x)$ acts as an identity function on the set of integers, on a set of functions of some type, or on any other kind of object. These functions are called *polymorphic* functions. The expression $\lambda x.x$ is same as $\lambda y.y$, called α-equivalent.

  The function $f : x \mapsto x^2$ is expressed as $\lambda x.x^2$, and $f(5)$ is $(\lambda x.x^2)(5) = 25$, called as β-reduction.

- Compute $x^2 + y^3$

$$\begin{aligned}
&(\lambda x.((\lambda y.x^2 + y^3)(2))(3)\\
&= (\lambda x.x^2 + 2^3)(3)\\
&= (3^2 + 2^3) = 17
\end{aligned}$$

- Compute $x^2 + y^3$, given a different λ expression.

$$\begin{aligned}
&(\lambda x.(\lambda y.x^2 + y^3))(2)(3)\\
&= (\lambda y.2^2 + y^3)(3)\\
&= (2^2 + 3^3) = 31
\end{aligned}$$

  In the above two examples, we note that it is the order of parentheses, which decides which λ argument is to be processed first. The normal processing is left to right, but it is the parentheses which decides processing order.

- The expression $\lambda n.(add\, n\, 1)$ denotes the successor function on the integers so that $((\lambda n.(add\, n\, 1))\, 5) = 6$. Note that "add" and 1 need to be predefined constants to define this function, and 5 must be predefined to apply the function as we have done.
- The abstraction $(\lambda f.(\lambda x.(f(f\ x))))$ describes a function with two arguments, a function and a value, that applies the function to the value twice. If *sqr* is the (predefined) integer function that squares its argument, then

$$\begin{aligned}(((\lambda f.(\lambda x.(f(f\ x))))\ sqr)\ 3) &= ((\lambda x.(sqr(sqr\ x)))3)\\ &= (sqr(sqr\ 3))\\ &= (sqr\ 9)\\ &= 81.\end{aligned}$$

  Here f is replaced by *sqr* and then x by 3.

The above examples show that the number of parentheses in lambda expressions can become quite large. The following notational conventions allow abbreviations that reduce the number of parentheses:

Uppercase letters and identifiers beginning with capital letters will be used as *metavariables* ranging over lambda expressions.

Application of the function associates to the left: $E_1\ E_2\ E_3$ means $((E_1\ E_2)\ E_3)$

The scope of "$\lambda < variable >$" in an abstraction extends as far to the right as possible, i.e.,

$$\lambda x.E_1\ E_2\ E_3$$

means

$$(\lambda x.(E_1\ E_2\ E_3))$$

and not

$$((\lambda x.E_1\ E_2)\ E_3).$$

So application of a function has a higher precedence than abstraction. Parentheses are needed for $(\lambda x.E_1\ E_2)\ E_3$, where E_3 is an argument to the function $\lambda x.E_1\ E_2$ and not as part of the body of the function.

An abstraction allows a list of variables that abbreviates a series of lambda abstractions, for example,

$$\lambda x\ y\ z.E$$

means

$$(\lambda x.(\lambda y.(\lambda z.E))).$$