Chapter 5
THE LAMBDA CALCULUS

Functions play a prominent role in describing the semantics of a programming language, since the meaning of a computer program can be considered as a function from input values to output values. In addition, functions play an essential role in mathematics, which means that much of the theory of functions, including the issue of computability, unfolded as part of mathematical logic before the advent of computers. In particular, 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.

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:

- 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.

In this chapter we take a brief but careful look at the lambda calculus, first defining it as a language and then viewing it as a computational formalism in light of its reduction rules. We end the chapter by implementing a lambda calculus evaluator in Prolog. In Chapter 10 we continue the study of functions with the goal of explaining recursive definitions.

5.1 CONCEPTS AND EXAMPLES

Our description of the lambda calculus begins with some motivation for the notation. A function is a mapping from the elements of a domain set to the elements of a codomain set given by a rule—for example,

$$\text{cube} : \text{Integer} \rightarrow \text{Integer} \quad \text{where } \text{cube}(n) = n^3.$$

Certain questions arise from such a function definition:

- What is the value of the identifier “cube”?
- How can we represent the object bound to “cube”?
- Can this function be defined without giving it a name?

Church’s lambda notation allows the definition of an anonymous function, that is, a function without a name:

$\lambda n \,.\, n^3$ 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:

$$(3,4) \mapsto 3^2+4 = 13 \quad \text{or} \quad (3,4) \mapsto 4^2+3 = 19.$$

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

$$\lambda n \,.\, \lambda m \,.\, n^2+m \quad \text{or} \quad \lambda m \,.\, \lambda n \,.\, n^2+m.$$

Most functional programming languages allow anonymous functions as values; for example, the function $\lambda n \,.\, n^3$ is represented as

(lambda (n) (* n n n)) in Scheme and

fn n => n * n * n in Standard ML.

Syntax of the 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 come 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).

The simplicity of lambda calculus syntax is apparent from a BNF specification of its concrete syntax:

```
<expression> ::= <variable>                          ; lowercase identifiers
               | <constant>                          ; predefined objects
               | ( <expression> <expression> )       ; combinations
               | ( λ <variable> . <expression> )     ; abstractions.
```

In the spirit of software engineering, 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. We will say more about the expressibility of the pure lambda calculus later in this chapter. For now, predefined constants will be allowed in our examples, including numerals (for example, 34), 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 expected that E_1 evaluates to a predefined function (a constant) or an abstraction, say (λx . 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 ope**rator** E_1 is called the **rator** and its argument or ope**rand** E_2 is called the **rand**. Formal definitions of free occurrences of variables and the replacement operation will be provided shortly.

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 λx . x denotes the identity function in the sense that ((λx . x) E) = E for any lambda expression E. Functions that allow arguments of many types, such as this identity function, are known as **polymorphic** operations. The lambda expression (λx . 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.

- The expression λn . (add n 1) denotes the successor function on the integers so that ((λn . (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

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

Here f is replaced by sqr and then x by 3. These 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:

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

2. Function application associates to the left.

   $E_1 E_2 E_3$ means $((E_1 E_2) E_3)$

3. The scope of "λ<variable>" in an abstraction extends as far to the right as possible.

   $\lambda x . E_1 E_2 E_3$ means $(\lambda x . (E_1 E_2 E_3))$ and not $((\lambda x . E_1 E_2) E_3)$.

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

4. An abstraction allows a list of variables that abbreviates a series of lambda abstractions.

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

5. Functions defined as lambda expression abstractions are anonymous, so the lambda expression itself denotes the function. As a notational convention, lambda expressions may be named using the syntax

   define <name> = <expression>.

   For example, given *define* Twice = λf . λx . f (f x), it follows that

   (Twice (λn . (add n 1)) 5) = 7.

   We follow a convention of starting defined names with uppercase letters to distinguish them from variables. Imagine that "Twice" is replaced by its definition as with a macro expansion before the lambda expression is reduced. Later we show a step-by-step reduction of this lambda expression to 7.

Example : Because of the sparse syntax of the lambda calculus, correctly parenthesizing (parsing) a lambda expression can be challenging. We illustrate the problem by grouping the terms in the following lambda expression:

(λn . λf . λx . f (n f x)) (λg . λy . g y).

We first identify the lambda abstractions, using the rule that the scope of lambda variable extends as far as possible. Observe that the lambda abstractions in the first term are ended by an existing set of parentheses.

(λx . f (n f x))	(λy . g y)
(λf . (λx . f (n f x)))	(λg . (λy . g y))
(λn . (λf . (λx . f (n f x))))	

Then grouping the combinations by associating them to the left yields the completely parenthesized expression:

((λn . (λf . (λx . (f ((n f) x))))) (λg . (λy . (g y)))). ∎

Curried Functions

Lambda calculus as described above seems to permit functions of a single variable only. The abstraction mechanism allows for only one parameter at a time. However, many useful functions, such as binary arithmetic operations, require more than one parameter; for example, sum(a,b) = a+b matches the syntactic specification sum : NxN → N, where N denotes the natural numbers.

Lambda calculus admits two solutions to this problem:

1. Allow ordered pairs as lambda expressions, say using the notation <x,y>, and define the addition function on pairs:

   sum <a,b> = a + b

   Pairs can be provided by using a predefined “cons” operation as in Lisp, or the pairing operation can be defined in terms of primitive lambda expressions in the pure lambda calculus, as will be seen later in the chapter.

2. Use the **curried** version of the function with the property that arguments are supplied one at a time:

add : N → N → N	(→ associates to the right)
where add a b = a + b	(function application associates to the left).

   Now (add a) : N → N is a function with the property that ((add a) b) = a + b. In this way, the successor function can be defined as (add 1).

Note that the notational conventions of associating → to the right and function application to the left agree. The terms “curried” and “currying” honor Haskell Curry who used the mechanism in his study of functions. Moses Schönfinkel actually developed the idea of currying before Curry, which means that it might more accurately be called “Schönfinkeling”.

The operations of currying and uncurrying a function can be expressed in the lambda calculus as

define Curry = λf . λx . λy . f <x,y>

define Uncurry = λf . λp . f (head p) (tail p)

provided the pairing operation <x,y> = (cons x y) and the functions (head p) and (tail p) are available, either as predefined functions or as functions defined in the pure lambda calculus, as we will see later.

Therefore the two versions of the addition operation are related as follows:

Curry sum = add and Uncurry add = sum.

One advantage of currying is that it permits the “partial application” of a function. Consider an example using Twice that takes advantage of the currying of functions:

define Twice = λf . λx . f (f x).

Note that Twice is another example of a polymorphic function in the sense that it may be applied to any function and element as long as that element is in the domain of the function and its image under the function is also in that domain. The mechanism that allows functions to be defined to work on a number of types of data is also known as **parametric polymorphism** .

If D is any domain, the syntax (or signature) for Twice can be described as

Twice : (D → D) → D → D.

Given the square function, sqr : N → N where N stands for the natural numbers, it follows that

(Twice sqr) : N → N

is itself a function. This new function can be named

define FourthPower = Twice sqr.

Observe that FourthPower is defined without any reference to its argument. Defining new functions in this way embodies the spirit of functional programming. Much of the power of a functional programming language lies in its ability to define and apply higher-order functions—that is, functions that take functions as arguments and/or return a function as their result. Twice is higher-order since it maps one function to another.

Semantics of Lambda Expressions

A lambda expression has as its meaning the lambda expression that results after all its function applications (combinations) are carried out. Evaluating a lambda expression is called **reduction**. The basic reduction rule involves substituting expressions for free variables in a manner similar to the way that the parameters in a function definition are passed as arguments in a function call. We start by defining the concepts of free occurrences of variables and the substitution of expressions for variables.

Definition: An *occurrence* of a variable v in a lambda expression is called **bound** if it is within the scope of a "λv"; otherwise it is called **free**. ∎

A variable may occur both bound and free in the same lambda expression; for example, in λx . y λy . y x the first occurrence of y is free and the other two are bound. Note the care we take in distinguishing between a variable and occurrences of that variable in a lambda expression.

The notation **E[v→E$_1$]** refers to the lambda expression obtained by replacing each free occurrence of the variable v in E by the lambda expression E$_1$. Such a substitution is called **valid** or **safe** if no free variable in E$_1$ becomes bound as a result of the substitution E[v→E$_1$]. An invalid substitution involves a **variable capture** or **name clash**.

For example, the naive substitution (λx . (mul y x))[y→x] to get (λx . (mul x x)) is unsafe since the result represents a squaring operation whereas the original lambda expression does not. We cannot allow the change in semantics engendered by this naive substitution, since we want to preserve the semantics of lambda expressions as we manipulate them.

The definition of substitution requires some care to avoid variable capture. We first need to identify the variables in E that are free by defining an operation that lists the variables that occur free in an expression. For example, *FV*(λx . y λy . y x z) = {y,z}.

Definition: The **set of free variables** (variables that occur free) in an expression E, denoted by ***FV*(E)**, is defined as follows:

a) *FV*(c) = ∅ for any constant c

b) *FV*(x) = {x} for any variable x

c) *FV*(E$_1$ E$_2$) = *FV*(E$_1$) ∪ *FV*(E$_2$)

d) *FV*(λx . E) = *FV*(E) – {x} ∎

A lambda expression E with no free variables (*FV*(E) = ∅) is called **closed**.

Definition : The **substitution** of an expression for a (free) variable in a lambda expression is denoted by **$E[v \rightarrow E_1]$** and is defined as follows:

a) $v[v \rightarrow E_1] = E_1$ for any variable v

b) $x[v \rightarrow E_1] = x$ for any variable $x \neq v$

c) $c[v \rightarrow E_1] = c$ for any constant c

d) $(E_{rator}\ E_{rand})[v \rightarrow E_1] = ((E_{rator}[v \rightarrow E_1])\ (E_{rand}[v \rightarrow E_1]))$

e) $(\lambda v\ .\ E)[v \rightarrow E_1] = (\lambda v\ .\ E)$

f) $(\lambda x\ .\ E)[v \rightarrow E_1] = \lambda x\ .\ (E[v \rightarrow E_1])$ when $x \neq v$ and $x \notin FV(E_1)$

g) $(\lambda x\ .\ E)[v \rightarrow E_1] = \lambda z\ .\ (E[x \rightarrow z][v \rightarrow E_1])$ when $x \neq v$ and $x \in FV(E_1)$,
where $z \neq v$ and $z \notin FV(E\ E_1)$ ∎

In part g) the first substitution $E[x \rightarrow z]$ replaces the bound variable x that will capture the free x's in E_1 by an entirely new bound variable z. Then the intended substitution can be performed safely.

Example : $(\lambda y\ .\ (\lambda f\ .\ f\ x)\ y)[x \rightarrow f\ y]$

$(\lambda y\ .\ (\lambda f\ .\ f\ x)\ y)[x \rightarrow f\ y]$

$= \lambda z\ .\ ((\lambda f\ .\ f\ x)\ z)[x \rightarrow f\ y]$	by g) since $y \in FV(f\ y)$
$= \lambda z\ .\ ((\lambda f\ .\ f\ x)[x \rightarrow f\ y]\ z[x \rightarrow f\ y])$	by d)
$= \lambda z\ .\ ((\lambda f\ .\ f\ x)[x \rightarrow f\ y]\ z)$	by b)
$= \lambda z\ .\ (\lambda g\ .\ (g\ x)[x \rightarrow f\ y])\ z$	by g) since $f \in FV(f\ y)$
$= \lambda z\ .\ (\lambda g\ .\ g\ (f\ y))\ z$	by d), b), and a) ∎

Observe that if $v \notin FV(E)$, then $E[v \rightarrow E_1]$ is essentially the same lambda expression as E; there may be a change of bound variables, but the structure of the expression remains unchanged.

The substitution operation provides the mechanism for implementing function application. In the next section we define the rules for simplifying lambda expressions based on this idea of substitution.

Exercises

1. Correctly parenthesize each of these lambda expressions:

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

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

c) $(\lambda f . \lambda y . \lambda z . f z y z) p x$

d) $\lambda x . x \lambda y . y \lambda z . z \lambda w . w z y x$

2. Find the set of free variables for each of the following lambda expressions:

   a) $\lambda x . x y \lambda z . x z$

   b) $(\lambda x . x y) \lambda z . w \lambda w . w z y x$

   c) $x \lambda z . x \lambda w . w z y$

   d) $\lambda x . x y \lambda x . y x$

3. Carry out the following substitutions:

   a) $(f (\lambda x . x y) \lambda z . x y z)[x \rightarrow g]$　　b) $(\lambda x . \lambda y . f x y)[y \rightarrow x]$

   c) $((\lambda x . f x) \lambda f . f x)[f \rightarrow g x]$　　d) $(\lambda f . \lambda y . f x y)[x \rightarrow f y]$

4. Using the function Twice and the successor function succ, define a function that

   a) adds four to its argument.

   b) adds sixteen to its argument.

5. Give a definition of the set of bound variables in a lambda expression E, denoted by *BV*(E).

6. Show by an example that substitutions can be carried out that alter the intended semantics if part g) of the substitution rule is replaced by:

   g') $(\lambda x . E)[v \rightarrow E_1] = \lambda z . (E[x \rightarrow z][v \rightarrow E_1])$ when $x \neq v$ and $x \in FV(E_1)$,

   where $z \notin FV(E\ E_1)$

5.2 LAMBDA REDUCTION

Simplifying or evaluating a lambda expression involves reducing the expression until no more reduction rules apply. The main rule for simplifying a lambda expression, called β-reduction, encompasses the operation of function application. Since substituting for free variables in an expression may cause variable capture, we first need a mechanism for changing the name of a bound variable in an expression—namely, α-reduction.

Definition: α-reduction

If v and w are variables and E is a lambda expression,

$$\lambda v \ .\ E \ \Rightarrow_{\alpha} \lambda w \ .\ E[v \rightarrow w]$$

provided that w does not occur at all in E, which makes the substitution E[v→w] safe. The equivalence of expressions under α-reduction is what makes part g) of the definition of substitution correct.

The α-reduction rule simply allows the changing of bound variables as long as there is no capture of a free variable occurrence. The two sides of the rule can be thought of as variants of each other, both members of an equivalence class of “congruent” lambda expressions. ∎

The example substitution at the end of the previous section contains two α-reductions:

$$\lambda y \ .\ (\lambda f \ .\ f\ x)\ y \ \Rightarrow_{\alpha} \lambda z \ .\ (\lambda f \ .\ f\ x)\ z$$

$$\lambda z \ .\ (\lambda f \ .\ f\ x)\ z \ \Rightarrow_{\alpha} \lambda z \ .\ (\lambda g \ .\ g\ x)\ z$$

Now that we have a justification of the substitution mechanism, the main simplification rule can be formally defined.

Definition: β-reduction

If v is a variable and E and E_1 are lambda expressions,

$$(\lambda v \ .\ E)\ E_1 \ \Rightarrow_{\beta} \ E[v \rightarrow E_1]$$

provided that the substitution $E[v \rightarrow E_1]$ is carried out according to the rules for a safe substitution.

This β-reduction rule describes the function application rule in which the actual parameter or argument E_1 is “passed to” the function (λv . E) by substituting the argument for the formal parameter v in the function. The left side (λv . E) E_1 of a β-reduction is called a **β-redex**—a term derived from the terms “reduction expression” and meaning an expression that can be β-reduced. β-reduction serves as the main rule of evaluation in the lambda calculus. α-reduction is simply used to make the substitutions for variables valid. ∎

The evaluation of a lambda expression consists of a series of β-reductions, possibly interspersed with α-reductions to change bound variables to avoid confusion. Take **E ⇒ F** to mean $E \Rightarrow_{\beta} F$ or $E \Rightarrow_{\alpha} F$ and let ⇒* be the reflexive and transitive closure of ⇒. That means for any expression E, E ⇒* E and for any three expressions, ($E_1 \Rightarrow^* E_2$ and $E_2 \Rightarrow^* E_3$) implies $E_1 \Rightarrow^* E_3$. The goal of evaluation in the lambda calculus is to reduce a lambda expression via ⇒ until it contains no more β-redexes.

To define an “equality” relation on lambda expressions, we also allow a β-reduction rule to work backward.

Definition : Reversing β-reduction produces the **β-abstraction** rule,

$$E[v \rightarrow E_1] \Rightarrow_\beta (\lambda v \, . \, E) \, E_1,$$

and the two rules taken together give **β-conversion** , denoted by $\Leftrightarrow_\beta$. Therefore $E \Leftrightarrow_\beta F$ if $E \Rightarrow_\beta F$ or $F \Rightarrow_\beta E$. Take $E \Leftrightarrow F$ to mean $E \Leftrightarrow_\beta F$, $E \Rightarrow_\alpha F$ or $F \Rightarrow_\alpha E$ and let $\Leftrightarrow$***** be the reflexive and transitive closure of $\Leftrightarrow$. Two lambda expressions E and F are **equivalent** or **equal** if $E \Leftrightarrow^* F$. ▮

We also allow reductions (both α and β) to subexpressions in a lambda expression. In particular, the following three rules expand the notion of reduction to components of combinations and abstractions:

1. $E_1 \Rightarrow E_2$ implies $E_1 \, E \Rightarrow E_2 \, E$.
2. $E_1 \Rightarrow E_2$ implies $E \, E_1 \Rightarrow E \, E_2$.
3. $E_1 \Rightarrow E_2$ implies $\lambda x \, . \, E_1 \Rightarrow \lambda x \, . \, E_2$.

A third rule, η-reduction, justifies an extensional view of functions; that is, two functions are equal if they produce the same values when given the same arguments. The rule is not strictly necessary for reducing lambda expressions and may cause problems in the presence of constants, but we include it for completeness.

Definition: η-reduction

If v is a variable, E is a lambda expression (denoting a function), and v has no free occurrence in E,

$$\lambda v \, . \, (E \; v) \Rightarrow_\eta E.$$ ▮

Note that in the pure lambda calculus every expression is a function, but the rule fails when E represents some constants; for example, if 5 is a predefined constant numeral, λx . (5 x) and 5 are not equivalent or even related.

However, if E stands for a predefined function, the rule remains valid as suggested by these examples:

$$\lambda x \, . \, (\text{sqr } x) \Rightarrow_\eta \text{sqr}$$

$$\lambda x \, . \, (\text{add } 5 \; x) \Rightarrow_\eta (\text{add } 5).$$

Remember, (add 5 x) abbreviates ((add 5) x).

The requirement that x should have no free occurrences in E is necessary to avoid a reduction such as

$$\lambda x \, . \, (\text{add } x \; x) \Rightarrow (\text{add } x),$$

which is clearly invalid.

Take $E \Leftrightarrow_\eta F$ to mean $E \Rightarrow_\eta F$ or $F \Rightarrow_\eta E$.

The η-reduction rule can be used to justify the extensionality of functions; namely, if f(x) = g(x) for all x, then f = g. In the framework of the lambda calculus, we can prove an extensionality property.

Extensionality Theor em: If F_1 x ⇒* E and F_2 x ⇒* E where x∉*FV*(F_1 F_2), then F_1 ⇔* F_2 where ⇔* includes η-reductions.

Proof: $F_1 \Leftrightarrow_\eta$ λx . (F_1 x) ⇔* λx . E ⇔* λx . (F_2 x) $\Leftrightarrow_\eta F_2$. ∎

Finally, in an applied lambda calculus containing predefined values and operations, we need a justification for evaluating combinations of constant objects. Such a rule is known as δ-reduction.

Definition: δ-reduction

If the lambda calculus has predefined constants (that is, if it is not pure), rules associated with those predefined values and functions are called δ rules; for example, (add 3 5) $\Rightarrow_\delta$ 8 and (not true) $\Rightarrow_\delta$ false. ∎

Example: Consider the following reduction of Twice (λn . (add n 1)) 5 where the leftmost β-redex is simplified in each step. For each β-reduction, the redex has its bound variable and argument highlighted in boldface.

Twice (λn . (add n 1)) 5 ⇒ (λ**f** . λx . (f (f x))) **(λn . (add n 1))** 5

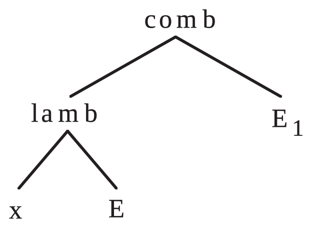
$\Rightarrow_\beta$ (λ**x** . ((λn . (add n 1)) ((λn . (add n 1)) x))) **5**

$\Rightarrow_\beta$ (λ**n** . (add n 1)) **((λn . (add n 1)) 5)**

$\Rightarrow_\beta$ (add ((λ**n** . (add n 1)) **5**) 1)

$\Rightarrow_\beta$ (add (add 5 1) 1) $\Rightarrow_\delta$ 7. ∎

The key to performing a reduction of a lambda calculus expression lies in recognizing a β-redex, (λv . E) E_1. Observe that such a lambda expression is a combination whose left subexpression is a lambda abstraction. In terms of the abstract syntax, we are looking for a structure with the form illustrated by the following tree:



Reduction Strategies

The main goal of manipulating a lambda expression is to reduce it to a "simplest form" and consider that as the value of the lambda expression.

Definition : A lambda expression is in **normal form** if it contains no β-redexes (and no δ-rules in an applied lambda calculus), so that it cannot be further reduced using the β-rule or the δ-rule. An expression in normal form has no more function applications to evaluate. ▮

The concepts of normal form and reduction strategy can be investigated by asking four questions:

1. Can every lambda expression be reduced to a normal form?
2. Is there more than one way to reduce a particular lambda expression?
3. If there is more than one reduction strategy, does each one lead to the same normal form expression?
4. Is there a reduction strategy that will guarantee that a normal form expression will be produced?

The first question has a negative answer, as shown by the lambda expression (λx . x x) (λx . x x). This β-redex reduces to itself, meaning that the only reduction path continues forever:

(λ**x** . x x) **(λx . x x)** $\Rightarrow_\beta$
(λ**x** . x x) **(λx . x x)** $\Rightarrow_\beta$ (λ**x** . x x) **(λx . x x)** $\Rightarrow_\beta$ …

The second question has an affirmative answer, as evidenced by the different reduction paths in the following examples.

Example : (λx . λy . (add y ((λz . (mul x z)) 3))) 7 5

Path 1 : (λ**x** . λy . (add y ((λz . (mul x z)) 3))) **7** 5
$\Rightarrow_\beta$ (λ**y** . (add y ((λz . (mul 7 z)) 3))) **5**
$\Rightarrow_\beta$ (add 5 ((λ**z** . (mul 7 z)) **3**))
$\Rightarrow_\beta$ (add 5 (**mul 7 3**)) $\Rightarrow_\delta$ (**add 5 21**) $\Rightarrow_\delta$ 26

Path 2 : (λx . λy . (add y ((λ**z** . (mul x z)) **3**))) 7 5
$\Rightarrow_\beta$ (λ**x** . λy . (add y (mul x 3))) **7** 5
$\Rightarrow_\beta$ (λ**y** . (add y (mul 7 3))) **5**
$\Rightarrow_\delta$ (λ**y** . (add y 21)) **5** $\Rightarrow_\beta$ (**add 5 21**) $\Rightarrow_\delta$ 26

In this example both paths lead to the same result, which is clearly in normal form. Note that in both paths λx must be reduced before λy because at this point in the reduction, λy is not part of a β-redex. ∎

Example : (λy . 5) ((λx . x x) (λx . x x))

Path 1 : (λ**y** . 5) **((λx . x x) (λx . x x))** $\Rightarrow_\beta$ 5

Path 2 : (λy . 5) ((λ**x** . x x) **(λx . x x)**)

$\Rightarrow_\beta$ (λy . 5) ((λ**x** . x x) **(λx . x x)**)

$\Rightarrow_\beta$ (λy . 5) ((λ**x** . x x) **(λx . x x)**) …

With this example the first path, which reduces the leftmost redex first, leads to a normal form expression, but the second path, which evaluates the rightmost application each time, results in a nonterminating calculation. ∎

These two reduction strategies have names.

Definition : A **normal order** reduction always reduces the leftmost outermost β-redex (or δ-redex) first. An **applicative order** reduction always reduces the leftmost innermost β-redex (or δ-redex) first. ∎

The operative words in this definition are “outermost” and “innermost”. Only when more than one outermost or innermost redex occur in a lambda expresion do we choose the leftmost of these redexes.

Definition : For any lambda expression of the form E = ((λx . B) A), we say that β-redex E is **outside** any β-redex that occurs in B or A and that these are **inside** E. A β-redex in a lambda expression is **outermost** if there is no β-redex outside of it, and it is **innermost** if there is no β-redex inside of it. ∎

Sometimes constructing an abstract syntax tree for the lambda expression can help show the pertinent β-redexes. For example, in Figure 5.1 the tree has the outermost and innermost β-redexes marked for the lambda expression (((λx . λy . (add x y)) ((λz . (succ z)) 5)) ((λw . (sqr w)) 7)). The two kinds of structured subexpressions are tagged by *comb* for combinations and *lamb* for lambda abstractions. From the tree, we can identify the leftmost outermost β-redex as ((λx . λy . (add x y)) ((λz . (succ z)) 5)) and the leftmost innermost as ((λz . (succ z)) 5). Remember that a β-redex (λv . E) E_1 is a combination consisting of an abstraction joined with another expression to be passed to the function. Some abstractions cannot be considered as innermost or outermost because they are not part of β-redexes.

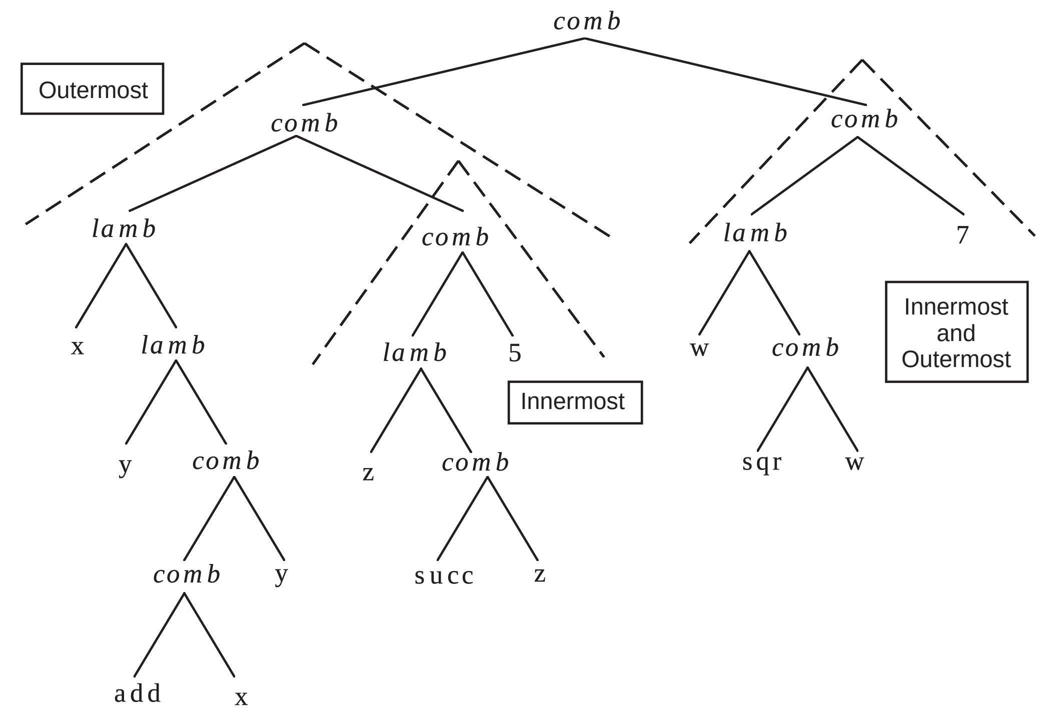


Figure 5.1: β-redexes in (((λx.λy.(add x y)) ((λz.(succ z)) 5)) ((λw.(sqr w)) 7))

When reducing lambda expressions in an applied lambda calculus, use a reduction strategy to decide when both β-reductions and δ-reductions are carried out. For predefined binary operations, we complete the δ-reduction only when both arguments have been evaluated. Strictly following an applicative order reduction strategy requires that ((λn . (add 5 n)) 8) be reduced to the lambda expression ((λn . add5 n) 8) where add5 : N → N is the function that adds 5 to its argument. To avoid making up unary functions from partially evaluated curried binary functions, we postpone the evaluation of add until both of its arguments are evaluated. So the reduction proceeds as follows:

$$((\lambda \mathbf{n}\ .\ (\text{add } 5\ \text{n}))\ \mathbf{8}) \Rightarrow_\beta (\mathbf{add\ 5\ 8}) \Rightarrow_\delta 13.$$

The two remaining questions can be answered by applying the results proved by Alonzo Church and J. Barkley Rosser in 1936.

Church-Rosser Theorem I: For any lambda expressions E, F, and G, if E ⇒* F and E ⇒* G, there is a lambda expression Z such that F ⇒* Z and G ⇒* Z.

Proof: The somewhat technical proof of this theorem can be found in [Barendregt84] or [MacLennan90]. ▮

Any relation that satisfies this condition is said to have the **diamond property** or the **confluence property**.The diagram below suggests the origin of these terms.

$$\begin{tikzcd}
& E \arrow[dl, Rightarrow, "*"'] \arrow[dr, Rightarrow, "*"] & \\
F \arrow[dr, Rightarrow, "*"] & & G \arrow[dl, Rightarrow, "*"'] \\
& Z &
\end{tikzcd}$$

Corollary: For any lambda expressions E, M, and N, if E ⇒* M and E ⇒* N where M and N are in normal form, M and N are variants of each other (equivalent with respect to α-reduction).

Proof: The only reduction possible for an expression in normal form is an α-reduction. Therefore the lambda expression Z in the theorem must be a variant of M and N by α-reduction only. ▮

This corollary states that if a lambda expression has a normal form, that normal form is unique up to a change of bound variables, thereby answering the third question.

Church-Rosser Theorem II: For any lambda expressions E and N, if E ⇒* N where N is in normal form, there is a normal order reduction from E to N.

Proof: Again see [Barendregt84]. ▮

The second Church-Rosser theorem answers the fourth question by stating that normal order reduction will produce a normal form lambda expression if one exists.

A normal order reduction can have either of the following outcomes:

1. It reaches a unique (up to α-conversion) normal form lambda expression.
2. It never terminates.

Unfortunately, there is no algorithmic way to determine for an arbitrary lambda expression which of these two outcomes will occur.

Turing machines are abstract machines designed in the 1930s by Alan Turing to model computable functions. It has been shown that the lambda calculus is equivalent to Turing machines in the sense that every lambda expression has an equivalent function defined by some Turing machine and vice versa. This equivalence gives credibility to Church's thesis.

Church's Thesis : The effectively computable functions on the positive integers are precisely those functions definable in the pure lambda calculus (and computable by Turing machines). ▮

The term “thesis” means a conjecture. In the case of Church’s thesis, the conjecture cannot be proved since the informal notion of “effectively computable function” is not defined precisely. But since all methods developed for computing functions have been proved to be no more powerful than the lambda calculus, it captures the idea of computable functions as well as we can hope.

Alan Turing proved a fundamental result, called the undecidability of the **halting problem**, which states that there is no algorithmic way to determine whether or not an arbitrary Turing machine will ever stop running. Therefore there are lambda expressions for which it cannot be determined whether a normal order reduction will ever terminate.

Correlation with Parameter Passing

The two fundamental reduction strategies, normal and applicative order, are related to techniques for passing parameters to a procedure or function in a programming language. Recall that an abstraction λx . B is an anonymous function whose formal parameter is x and whose body is the lambda expression B.

1. **Call by name** is the same as normal order reduction except that no redex in a lambda expression that lies within an abstraction (within a function body) is reduced. With call by name, an actual parameter is passed as an unevaluated expression that is evaluated in the body of the function being executed each time the corresponding formal parameter is referenced. Normal order is ensured by choosing the leftmost redex, which will always be an outermost one with an unevaluated operand.

2. **Call by value** is the same as applicative order reduction except that no redex in a lambda expression that lies within an abstraction is reduced. This restriction corresponds to the principle that the body of a function is not evaluated until the function is called (in a β-reduction). Applicative order means the argument to a function is evaluated before the function is applied.

Example: The call by value reduction of

(λx . (λf . f (succ x)) (λz . (λg . (λy . (add (mul (g y) x)) z)))) ((λz . (add z 3)) 5)

proceeds as follows:

(λx . (λf . f (succ x)) (λz . (λg . (λy . (add (mul (g y) x))) z))) ((λ**z** . (add z 3)) **5**)
$\Rightarrow_\beta$ (λx . (λf . f (succ x)) (λz . (λg . (λy . (add (mul (g y) x))) z))) (**add 5 3**)
$\Rightarrow_\delta$ (λ**x** . (λf . f (succ x)) (λz . (λg . (λy . (add (mul (g y) x))) z))) **8**
$\Rightarrow_\beta$ (λ**f** . f (succ 8)) **(λz . (λg . (λy . (add (mul (g y) 8))) z))**
$\Rightarrow_\beta$ (λz . (λg . (λy . (add (mul (g y) 8))) z)) (**succ 8**)
$\Rightarrow_\delta$ (λ**z** . (λg . (λy . (add (mul (g y) 8))) z)) **9**
$\Rightarrow_\beta$ (λg . (λy . (add (mul (g y) 8))) 9) ∎

In this example, the reduction of the argument, ((λz . (add z 3)) 5), can be thought of as an optimization, since we pass in a value and not an unevaluated expression that would be evaluated twice in the body of the function. Observe that the final result stops short of normal form with call by value semantics.

Constants in the Pure Lambda Calculus

If Church's thesis is to be credible, there must be a way to define the nonnegative integers in the pure lambda calculus. Furthermore, to allow the definition of uncurried binary functions, we need to be able to define a list constructor and list selectors. Since so many functions employ conditional definitions, we also need to define Boolean constants and operations. Although all of these constants can be defined as functions in the pure lambda calculus, we consider only a few examples here.

The function "Pair" encapsulates two values in a given order; it is essentially the dotted pair notion (cons) in Lisp.

define Pair = λa . λb . λf . f a b

Two selector functions "Head" and "Tail" confirm that Pair implements the cons operation.

define Head = λg . g (λa . λb . a)

define Tail = λg . g (λa . λb . b)

Now the correctness of the definitions is verified by reductions:

Head (Pair p q) $\Rightarrow$ (λ**g** . g (λa . λb . a)) **((λa . λb . λf . f a b) p q)**
$\Rightarrow_\beta$ ((λ**a** . λb . λf . f a b) **p** q) (λa . λb . a)
$\Rightarrow_\beta$ ((λ**b** . λf . f p b) **q**) (λa . λb . a)
$\Rightarrow_\beta$ (λ**f** . f p q) **(λa . λb . a)**
$\Rightarrow_\beta$ (λ**a** . λb . a) **p** q $\Rightarrow_\beta$ (λ**b** . p) **q** $\Rightarrow_\beta$ p

As with “cons” in Lisp, “Pair” is sufficient to construct lists of arbitrary length; for example, assuming we have positive integers and a special constant,

define Nil = λx . λa . λb . a,

a list can be constructed as follows:

define [1, 2, 3, 4] = Pair 1 (Pair 2 (Pair 3 (Pair 4 Nil))).

Combinations of the selector functions can choose items from the list:

Head (Tail (Tail [1, 2, 3, 4])) $\Rightarrow$ 3.

Several ways have been proposed for representing the natural numbers in the pure lambda calculus. In each case, cardinal values are encoded as patterns in function definitions. Our approach will be to code a natural number by the number of times a function parameter is applied:

define 0 = λf . λx . x

define 1 = λf . λx . f x

define 2 = λf . λx . f (f x)

define 3 = λf . λx . f (f (f x)) and so on.

Based on this definition of numerals, the standard arithmetic operations can be defined as functions in the lambda calculus. We give two examples here.

Successor function, Succ : N $\rightarrow$ N

define Succ = λn . λf . λx . f (n f x)

Addition operation, Add : N $\rightarrow$ N $\rightarrow$ N

define Add = λm . λn . λf . λx . m f (n f x).

As an example of a reduction using this arithmetic of pure lambda expressions, consider the successor of 2. The bound variables are altered in 2 to make the reduction easier to follow, but the changes are really unnecessary.

$$\begin{aligned}
\text{Succ } 2 &\Rightarrow (\lambda \mathbf{n} . \lambda f . \lambda x . f\,(n\,f\,x))\,\mathbf{(\lambda g . \lambda y . g\,(g\,y))} \\
&\Rightarrow_\beta \lambda f . \lambda x . f\,((\lambda \mathbf{g} . \lambda y . g\,(g\,y))\,\mathbf{f}\,x) \\
&\Rightarrow_\beta \lambda f . \lambda x . f\,((\lambda \mathbf{y} . f\,(f\,y))\,\mathbf{x}) \\
&\Rightarrow_\beta \lambda f . \lambda x . f\,(f\,(f\,x)) = 3
\end{aligned}$$

Functional Programming Languages

All functional programming languages can be viewed as syntactic variations of the lambda calculus. Certainly the fundamental operations in all functional languages are the creation of functions—namely, lambda abstraction—and the application of functions, the two basic operations in the lambda calculus. The naming of functions or other objects is pure syntactic sugar since the values named can be substituted directly for their identifiers, at least for nonrecursive definitions. Recursive definitions can be made nonrecursive using a fixed point finder, an issue that we cover in Chapter 10.

Most current functional programming languages follow static scoping to resolve references to identifiers not declared in a local scope. Static or lexical scoping means that nonlocal variables refer to bindings created in the textually enclosing blocks. In contrast, with dynamic scoping nonlocal references are resolved by following the calling chain. We examine these two possibilities more carefully in Chapters 6 and 8.

Nested scopes are naturally created by the activations of nested function calls where the formal parameters designate the local identifiers. A **let** expression is an alternate means of writing a lambda expression that is a function application.

> **let** x=5 **in** (add x 3) means (λx . (add x 3)) 5.

In some functional languages, a **where** expression acts as an alternative to the let expression.

> (add x 3) **where** x=5 also means (λx . (add x 3)) 5.

A recursive let or where, sometimes called "letrec" or "whererec", requires the fixed point operator, but we postpone a discussion of that until Chapter 10. At any rate, the syntactic forms of a functional programming language can be translated into the lambda calculus and studied in that context. For more on this translation process, see the further readings at the end of this chapter.

Exercises

1. Identify the innermost and outermost β-redexes in the following lambda expression and draw its abstract syntax tree:

   (λx y z . (add x (mul y z))) ((λx . (succ x)) 5) 12 ((λw . (w 4)) sqr)

2. Use both normal order reduction and applicative order reduction to reduce the following lambda expressions. Reach a normal form representation if possible.

   a) (λg . g 5) (λx . (add x 3))

   b) (λx . (λy z . z y) x) p (λx . x)

   c) (λx . x x x) (λx . x x x)

   d) (λx . λy . (add x ((λx . (sub x 3)) y))) 5 6

   e) (λc . c (λa . λb . b)) ((λa . λb . λf . f a b) p q)

   f) Twice (λn . (mul 2 (add n 1))) 5

   g) Twice (Twice (λn . (mul 2 (add n 1)))) 5

   h) Twice Twice sqr 2

   i) (λx . ((λz . (add x x)) ((λx . λz . (z 13 x)) 0 div))) ((λx . (x 5)) sqr)

3. Use call by value semantics to reduce the following lambda expressions:

   a) (λf . f add (f mul (f add 5))) (λg . λx . g x x)

   b) (λx . λf . f (f x)) ((λy . (add y 2)) ((λz . (sqr z)) ((λy . (succ y)) 1))) sqr

4. Show that Tail (Pair p q) $\Rightarrow_\beta$ q.

5. Using constants defined in the *pure* lambda calculus, verify the following reductions:

   a) Succ 0 ⇒ 1

   b) Add 2 3 ⇒ 5

6. Using the definitions of Pair, Head, Tail, Curry, and Uncurry, where

   define Curry = λf . λx . λy . f (Pair x y)

   define Uncurry = λf . λp . f (Head p) (Tail p)

   carry out the following reductions:

   a) Curry (Uncurry h) ⇒ h

   b) Uncurry (Curry h) (Pair r s) ⇒ h (Pair r s)

7. Translate these “let” expressions into lambda expressions and reduce them. Also write the expressions using “where” instead of “let”.

   a) **let** x = 5 **in let** y = (add x 3) **in** (mul x y)

   b) **let** a = 7 **in let** g = λx . (mul a x) **in let** a = 2 **in** (g 10)

5.3 LABORATORY: A LAMBDA CALCULUS EVALUATOR

In this section we implement a lambda calculus evaluator in Prolog for an applied lambda calculus with a few constants. Extensions are suggested in the exercises. The implementation was inspired by a lambda calculus evaluator written in Lisp and described in [Gordon88].

The evaluator expects to read a file containing one lambda expression to be reduced according to a normal order reduction strategy. A sample execution follows:

```
>>> Evaluating an Applied Lambda Calculus <<<
Enter name of source file: twice
    ((L f x (f (f x))) (L n (mul 2 (add n 1))) 5)
Successful Scan
Successful Parse
Result =  26
yes
```

Since Greek letters are missing from the ascii character set, we use "L" to stand for λ in lambda expressions. The concrete syntax recognized by our implementation, displayed in Figure 5.2, allows only two abbreviations for eliminating parentheses:

(L x y E) means (L x (L y E)), which stands for λx . λy . E and

(E_1 E_2 E_3) means ((E_1 E_2) E_3).

In particular, outermost parentheses are never omitted.

```
<expression> ::= <identifier> | <constant>
                   | ( L <identifier>+ <expression> )
                   | ( <expression>+ <expression> )
<constant> ::= <numeral> | true | false | succ | sqr
                   | add | sub | mul
```

Figure 5.2: Concrete Syntax for an Applied Lambda Calculus

Scanner and Parser

The scanner for the implementation needs to recognize identifiers starting with lowercase letters, numerals (only nonnegative to simplify the example), parentheses, and the reserved words "L", "true", and "false", together with the identifiers for the predefined operations. The scanner for Wren can be adapted to recognize these tokens for the lambda calculus. Remember that an atom starting with an uppercase letter, such as L, must be written within apostrophes in Prolog.

The parser takes the list of tokens Tokens from the scanner and constructs an abstract syntax tree for the lambda expression as a Prolog structure. The examples below illustrate the tags used by the abstract syntax.

Lambda Expression:	λx . (sqr x)
Concrete Syntax:	(L x (sqr x))
Abstract Syntax Tree:	*lamb*(x,*comb*(*con*(sqr),*var*(x)))
Lambda Expression:	(λx . λy . (add x y)) 5 8
Concrete Syntax:	((L x y (add x y)) 5 8)
Abstract Syntax Tree:	*comb*(*comb*(*lamb*(x,*lamb*(y,*comb*(*comb*(*con*(add), *var*(x)),*var*(y)))),*con*(5)),*con*(8))

The abstract syntax tree for the second example is displayed in Figure 5.3.

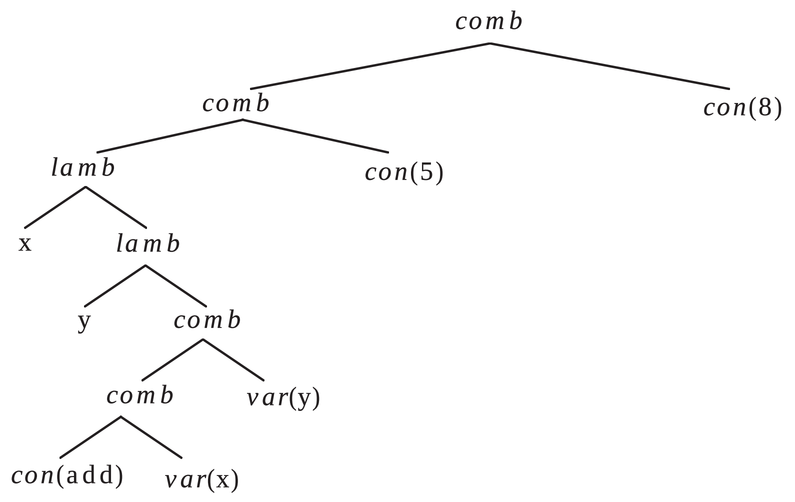


Figure 5.3: Abstract Syntax Tree for (λx . λy . (add x y)) 5 8

Figure 5.4 gives the abstract syntax used by the lambda calculus evaluator following a tagged structure format. Identifiers and constants are left unspecified since they are handled by the scanner.

```
Expression ::= var(Identifier) | con(Constant)
             | lamb(Identifier,Expression)
             | comb(Expression,Expression)
```

Figure 5.4: Abstract Syntax for an Applied Lambda Calculus

We invoke the parser by means of program(expr(E),Tokens,[eop]). Following the concrete syntax, we obtain the logic grammar shown in Figure 5.5 that recognizes variables, constants, lambda abstractions, and combinations.

```
program(expr(E)) --> expr(E).

expr(lamb(X,E)) --> [lparen],['L'],[var(X)],expr(E1),restlamb(E1,E).
    restlamb(E,E) --> [rparen].
    restlamb(var(Y),lamb(Y,E)) --> expr(E1),restlamb(E1,E).

expr(E) --> [lparen],expr(E1),expr(E2),restcomb(E1,E2,E).
    restcomb(E1,E2,comb(E1,E2)) --> [rparen].
    restcomb(E1,E2,E) --> expr(E3), restcomb(comb(E1,E2),E3,E).

expr(var(X)) --> [var(X)].         expr(con(N)) --> [num(N)].
expr(con(true)) --> [true].        expr(con(false)) --> [false].
expr(con(succ)) --> [succ].        expr(con(sqr)) --> [sqr].
expr(con(add)) --> [add].          expr(con(sub)) --> [sub].
expr(con(mul)) --> [mul].
```

Figure 5.5: Logic Grammar for the Lambda Calculus

The Lambda Calculus Evaluator

The evaluator proceeds by reducing the given lambda expression until no further reduction is possible. That reduction can be carried out in Prolog using

```
evaluate(E,NewE) :- reduce(E,TempE), evaluate(TempE,NewE).
evaluate(E,E).
```

When reduce fails, the second clause terminates the evaluation, returning the current lambda expression.

Before we can describe the reduction predicate reduce, which encompasses the β-reduction rule, the δ-reduction rule, and the overall normal order reduction strategy, we need to define the substitution operation used by β-reduction. As a first step, we formulate a predicate for determining the free variables in a lambda expression:

```
freevars(var(X),[X]).

freevars(con(C),[ ]).

freevars(comb(Rator,Rand),FV) :-  freevars(Rator,RatorFV),
                                  freevars(Rand,RandFV),
                                  union(RatorFV,RandFV,FV).

freevars(lamb(X,E),FV) :- freevars(E,F), delete(X,F,FV).
```

The utility predicates union(S1,S2,S3), forming S3 as the union of lists S1 and S2, and delete(X,S1,S2), which removes the item X from the list S1 producing S2, are left for the reader to supply. Compare the Prolog definition of freevars with the specification of *FV*(E) in section 5.1.

The substitution predicate follows the definition in section 5.1 case by case, except that parts f) and g) combine into one clause with a guard, member(X,F1), where F1 holds the free variables in E1, to distinguish the cases. If the guard fails, we are in part f) and just carry out the substitution in E. If, however, X is a free variable in E1, we construct a new variable Z using the predicate variant and perform the two substitutions shown in the definition of part g).

```
subst(var(V),V,E1,E1).                                          % a)

subst(var(X),V,E1,var(X)).                                      % b)

subst(con(C),V,E1,con(C)).                                      % c)

subst(comb(Rator,Rand),V,E1,comb(NewRator,NewRand)) :-          % d)
                                    subst(Rator,V,E1,NewRator),
                                    subst(Rand,V,E1,NewRand).

subst(lamb(V,E),V,E1,lamb(V,E)).                                % e)

subst(lamb(X,E),V,E1,lamb(Z,NewE)) :- freevars(E1,F1),
               (member(X,F1), freevars(E,F),                    % g)
                              union(F,[V],F2), union(F1,F2,FV),
                              variant(X,FV,Z),
                              subst(E,X,var(Z),TempE),
                              subst(TempE,V,E1,NewE)
                            ; subst(E,V,E1,NewE), Z=X) .        % f)
```

The predicate variant(X,L,NewX) builds a variable that is different from all the variables in the list L by adding apostrophes to the end of the variable bound to X.

```
variant(X,L,NewX) :- member(X,L),prime(X,PrimeX),variant(PrimeX,L,NewX).
variant(X,L,X).
prime(X,PrimeX) :- name(X,L), concat(L,[39],NewL), name(PrimeX,NewL).
```

The ascii value 39 indicates an apostrophe. The reader needs to furnish the utility predicates member and concat to finish the specification of the substitution operation. See Appendix A for definitions of these predicates.

The reduce predicate performs a one-step reduction of a lambda expression, using pattern matching to provide a normal order reduction strategy. Since no clauses match a variable or a constant, no reduction exists for them—they are already in normal form. The first clause handles a β-reduction because the pattern is the outermost β-redex. The second clause executes a predefined function (δ-reduction) by calling a predicate compute to carry out the arithmetic and Boolean operations. The third and fourth clauses reduce the rator and rand expressions in that order, thus ensuring the evaluation of outermost β-redexes from left to right. Finally, the last clause simplifies a lambda expression by reducing its body.

```
reduce(comb(lamb(X,Body),Arg),R) :- subst(Body,X,Arg,R).              % 1
reduce(comb(con(C),con(Arg)),R) :- compute(C,Arg,R).                  % 2
reduce(comb(Rator,Rand),comb(NewRator,Rand)) :-
                                          reduce(Rator,NewRator).     % 3
reduce(comb(Rator,Rand),comb(Rator,NewRand)) :-
                                          reduce(Rand,NewRand).       % 4
reduce(lamb(X,Body),lamb(X,NewBody)) :- reduce(Body,NewBody).         % 5
```

The compute predicate evaluates the arithmetic operations using Prolog's native numerical operations. We give it the responsibility for attaching the con tag to the result because some predefined operations may not need a tag. An exercise asks the reader to implement an "if" operation, which produces an untagged answer since it only evaluates one of its branches.

```
compute(succ,N,con(R)) :- R is N+1.
compute(sqr,N,con(R)) :- R is N*N.
compute(add,N,con(add(N))).      compute(add(M),N,con(R)) :- R is M+N.
compute(sub,N,con(sub(N))).      compute(sub(M),N,con(R)) :- R is M-N.
compute(mul,N,con(mul(N))).      compute(mul(M),N,con(R)) :- R is M*N.
```

Notice how the curried binary operations are handled by constructing a Prolog term containing the left operand, tagged by the operation name to represent the partially evaluated operation. To compute (add 2 3), which has as its abstract syntax tree

```
comb(comb(con(add),con(2)),con(3)),
```

the evaluation proceeds as follows using add(2) to represent the partially applied operation:

```
comb(comb(con(add),con(2)),con(3))  ⇒  comb(con(add(2)),con(3))
                                    ⇒  con(5).
```

The final result can be printed by a pretty-printer predicate pp. To visualize the progress of the reduction, insert pp(TempE) in the evaluate predicate.

```
pp(var(X)) :- write(X).

pp(con(C)) :- write(C).

pp(lamb(X,E)) :- write('(L '),write(X),tab(1),pp(E),write(')').

pp(comb(Rator,Rand)) :- write('('),pp(Rator),tab(1),pp(Rand),write(')').
```

If the parser produces a structure of the form expr(Exp), the lambda calculus evaluator can be invoked using the query

```
evaluate(Exp,Result), nl, write('Result = '), pp(Result), nl.
```

Although the lambda calculus evaluator in Prolog suffers from a lack of efficiency, it provides an effective tool for describing the reduction of lambda expressions. The reader will find that the task of matching parentheses correctly in an expression inflicts the most discomfort to a user of the evaluator.

Exercises

1. Complete the Prolog code for the lambda calculus evaluator by writing the missing utility predicates and test the evaluator on some of the lambda expressions in section 5.2.

2. Define a Prolog predicate boundvars(E,List) that produces a list containing the bound variables in E.

3. Add the operations “div”, “pred” (for predecessor), “and”, “or”, “not”, “zerop” (testing whether an expression equals zero), and “(if E_1,E_2,E_3)” to the evaluator. Test the evaluator on the lambda expression:

   ((L x (if (zerop x) 5 (div 100 x))) 0).

4. Add lists of arbitrary lambda expressions “[E_1, E_2, …, E_n]”, the operations “cons”, “head”, “tail”, and “nullp” (testing whether a list is empty), and the constant “nil” to the evaluator.

5. Change the evaluator to applicative order reduction and test it by comparing it to the normal order reducer.

6. Add a mechanism to the evaluator for giving definitions prior to the lambda expression to be evaluated. An example program is

```
define Twice = (L f x (f (f x)))
define Thrice = (L f x (f (f (f x))))
define Double = (L x (add x x))
(Thrice Twice Double 3).
```

   Provide a predicate elaborate that enters the definitions into an environment structure, say

```
env(Double,lamb(x,comb(comb(con(add),var(x)),var(x))),
    env(Thrice,lamb(f,lamb(x,comb(var(f),comb(var(f),comb(var(f),var(x)))))),
        env(Twice,lamb(f,lamb(x,comb(var(f),comb(var(f),var(x))))),nil)))
```

   where nil represents an empty environment, and a predicate expand that replaces the defined names by their bindings in the environment. Design the mechanism so that definitions may refer to names that have already been defined in the list of declarations. This modification will require changes to the scanner and parser as well as the evaluator.

5.4 FURTHER READING

Many books on functional programming contain material on the lambda calculus, including [Michaelson89], [Revesz88], [Field88], and [Reade89]. For a short but very clear presentation of the lambda calculus, we recommend [Gordon88], which contains a lambda calculus evaluator written in Lisp. The text by Bruce MacLennan [MacLennan90] contains readable proofs for some of the theoretical results that we skipped in this chapter. For an advanced and exhaustive look at the lambda calculus, see [Barendregt84].

Several books describe a methodology for translating programs in a functional language into the lambda calculus, among them [Peyton Jones87] and [Diller88].