

COMPILING LAMBDA CALCULUS

Nils M Holm

Contents

Preface	10
Lambda Calculus	13
Notation	13
<i>Syntax and Short Cuts</i>	15
Free and Bound Variables	15
<i>Substitution</i>	17
Conversion and Reduction	18
<i>Convertibility</i>	20
<i>Redexes and Normal Forms</i>	20
<i>Reduction Strategies</i>	21
<i>Confluence</i>	23
<i>Extensionality</i>	24
<i>Variable Convention</i>	25
Application	27
Church Numerals	27
<i>Basic Operations</i>	28
<i>Predecessor and Difference</i>	29
Propositional Functions	31
Fixed Points and Recursion	33
<i>Recursion</i>	34
<i>Schematic Definitions</i>	37
Ordered Pairs	38
<i>Lists</i>	39
<i>Typed Pairs</i>	40
Compilation	45
The Language	45
Abstract Reduction Machine	46
<i>A Naive Compiler</i>	47
<i>Bindings and Free Variables</i>	48
<i>Abstract Interpretation</i>	50

Environment Propagation	57
<i>Replacing Free Variables with Indexes</i>	58
<i>Environment Vector Compilation</i>	61
<i>Interpretation of Environment Vectors</i>	63
Inlining Primitive Functions	67
Mutable Variables	71
<i>Mutation and Environment Vectors</i>	73
<i>Mutation and Recursion</i>	77
Multi-Variable Functions	79
Variadic Functions	83
Error Handling	85
Unrolling the Loops	89
<i>Call Frames</i>	90
<i>The Virtual Machine</i>	90
<i>VM Instructions</i>	92
<i>Unrolling the Interpreter Loops</i>	96
<i>The Compiler</i>	96
Emitting Native Code	101
<i>The Compiler</i>	102
<i>Runtime Support</i>	108
<i>Sample Run</i>	117
An LC-Based Language	121
Definition	122
Implementation	123
<i>Runtime Support</i>	134
Optimizations	136
<i>Environment Pruning</i>	137
<i>Elimination of Boxing</i>	138
<i>Elimination of Saved Variables</i>	139
Extension	140
<i>Expanding Schematic Definitions</i>	141
<i>Global Definitions</i>	144

Appendix	149
Finding Bound Variables	149
Finding Free Variables	149
Renaming Variables	150
Lambda Calculator	151
Scheme VM	152
Mathematical Symbols	158
List of Figures	159
List of Definitions	160
Bibliography	162
Index	164

Preface

Lambda calculus is an abstract term rewriting system that was originally designed to solve mathematical problems on a sheet of paper. Paper is an inherently *immutable* tool, though: each term is rewritten by placing a modified term under the original one.

Computer systems, on the other hand, are inherently mutable systems: problems are solved by changing the values of registers.

This book describes how to adapt lambda calculus to computer-based environments, by interpreting it or translating it to a notation that is better fitted for processing by a register-based machine.

Chapter one of the book describes lambda calculus in general: its syntax, composition of formulae, common abbreviations, and the rules that are used to *reduce* (i.e. compute) lambda terms. This chapter introduces the basic theory of lambda calculus.

The second chapter illustrates how lambda calculus works in practice, by applying it to fundamental problems like the implementation of natural numbers, ordered pairs, and lists. It also introduces fixed points and recursive formulae.

The third chapter implements a very simple subset of the Scheme language, which is loosely based on lambda calculus.

It starts with simple interpreter for a tiny subset language and then modifies it subsequently in order to cover a wider range of the Scheme language and finally shifts the focus from interpretation to compilation and code generation. The full code for all interpreters and compilers is included in the text.

The concluding chapter presents a language that resembles lambda calculus more closely, by implementing currying and partial evaluation. The implications of the design are outlined and compared to typical LISP and Scheme implementations. Again, a working compiler for the language is included in the text.

This book is intended for compiler writers and people who are interested in lambda calculus and LISP languages. It focuses on theory as well as interpretation and code generation.

The code in the book is mostly in lambda calculus and Scheme, with a few short passages in C for addressing the more technical side of program interpretation (i.e. runtime support and code generation).

Machine-readable versions of the programs in this book may be found on the author's homepage: <http://t3x.org>.

Although the code in the book is simple and straight-forward, some familiarity with Scheme and C is certainly beneficial to following the text, especially the later chapters.

And now, as usual, enjoy the tour!

Nils M Holm, Nov 2016

Lambda Calculus

Lambda calculus (or λ -calculus) forms the theoretical basis for most functional programming languages. It is often associated with concepts like “anonymous functions” or “closures” but we will start from the beginning here and have a closer look at the formal system that is λ -calculus.

A *formal system* is a mathematical concept for manipulating terms of a formal language. It is comprised of three parts:

- notation (syntax)
- theory (axioms and rules)
- semantics (meaning of terms)

In the following, the notation and theory of λ -calculus will be given formally and semantics will be specified informally as required.

Notation

The *alphabet* of λ -calculus (LC) is the following:

$$\lambda \ (\) \ a_i \cdots z_i \ A_i \cdots Z_i$$

where $a \cdots z$ denotes the letters from a to z , a_i denotes the letters a_0, a_1, \dots , etc. Any letter of the form x_0 is normally written x . There is an infinite number of letters. Some texts use $\bar{a}, \bar{\bar{a}}$, etc instead of a_1, a_2, \dots .

By convention, lower case letters (x, y, \dots) represent variables and upper case letters (M, N, \dots) represent terms, which may in turn be variables, abstractions, or applications (see below).

A *well-formed formula* (*wff*) is a any combination of symbols from the alphabet of a formal system that forms a correct term under the rules of that system. The wffs of LC are defined inductively as follows (see [Church]):

(W1) $x \in \{a_i, \dots, z_i, A_i, \dots, Z_i\}$ is a wff. (variable)

(W2) If x is a variable and M is a wff, $(\lambda x M)$ is a wff. (abstraction)

(W3) if M and N are wffs, (MN) is a wff. (application)

Only combinations of symbols that can be built from the above rules are valid terms of the LC. For instance:

$x \ (\lambda xx) \ (MN) \ (x(yz)) \ ((\lambda x(xx))y) \ (\lambda a(\lambda b((Fb)a)))$

A *variable* is similar to a variable in a programming language, with one minor difference: a variable does not necessarily have a value, but it can stand for itself. In other words: LC does not assign any special meaning to a variable other than its lexical form. Variables may be acted upon by the rules outlined later in this chapter.

Abstraction can be thought of as the creation of a new *function* $(\lambda x M)$, where x is the independent variable of the function and M is its term. In pure λ -calculus, x must appear in M , while in λK -calculus (pg 22), it does not have to. This text deals with λK -calculus exclusively.

(FN) denotes the *application* of the term F to the term N . If F is an abstraction of the form $(\lambda x M)$, then

(App) $(FN) = ((\lambda x M)N) = M[x:=N]$

where $M[x:=N]$ denotes M with all occurrences of x replaced by N , i.e. the *substitution* of N for x in M . For example:

$((\lambda x(\lambda y(yx)))A) = (\lambda y(yx))[x:=A] = (\lambda y(yA))$

The F in (FN) is called the *function* and N is called the *argument* in the context of application.

If M in (MN) is not an abstraction, (MN) is in its “normal form” (pg 20) and the term is not rewritten in any way.

Unfortunately, application is a bit more complex than outlined here. We will return to it shortly (pg 17).

Syntax and Short Cuts

Because λ -terms can become pretty unwieldy pretty quickly, the *abbreviations* (A1) through (A5), listed in figure 1, are commonly used.

(A1)	(λxM)	λxM
(A2)	(MN)	MN
(A3)	$((MN)P)$	MNP
(A4)	$(M(NP))$	$M(NP)$
(A5)	$(\lambda x(\lambda yM))$	$\lambda xy(M)$

Figure 1 - Abbreviations

Furthermore, a dot is used to bracket everything to the right of the dot, e.g.:

$$\lambda x. yz \equiv \lambda x(yz) \equiv (\lambda x(yz))$$

$$\lambda xyz. xz(yz) \equiv \lambda xyz(xz(yz)) \equiv (\lambda x(\lambda y(\lambda z((xz)(yz))))))$$

The notation $x \equiv y$ is used to indicate that the terms x and y are syntactically *equivalent*. The notation is also used to *define* new functions, by assigning names to terms.

As can be seen from (A3), application *associates* to the left, i.e. in a sequence of two operations, the left one will take precedence over the right one: $MNP = (MN)P$.

Abstraction, on the other hand, associates to the right (A5):

$$\lambda xyz. xz(yz) = \lambda x(\lambda y(\lambda z. xz(yz)))$$

This follows immediately from definition (W2) of wffs (pg 13).

Free and Bound Variables

Given the term $(\lambda x\lambda y. xy)y$, the result of a naive substitution as described previously (pg 14) would result in the term $\lambda y. yy$, which is clearly different from $\lambda y. xy$. The problem in this case is that the variable y was *captured* during substitution, i.e. it was substituted in a term in which it was already “bound”.

A variable x is *bound* in a term M , if M contains at least one *sub-term* of the form λxN . A sub-term can be:

- a variable
- the M in λxM
- either the N or M in MN

A variable that is not bound in a term is called *free* in that term.

For example:

The variable x is free in the term x .

Both x and y are *free* in xy .

x is bound and y is free in $\lambda x. xy$.

x is bound in $x\lambda xx$ (because it is bound in λxx).

In general, the set of bound variables $\underline{BV}(M)$ of a term M is defined as follows.

(The notation \underline{XY} is used to denote symbolic names consisting of multiple characters; the bar is used to distinguish the name “XY” from the term XY (X applied to Y).)

$$(BV1) \quad \underline{BV}(x) = \emptyset$$

$$(BV2) \quad \underline{BV}(\lambda xM) = \{x\} \cup \underline{BV}(M)$$

$$(BV3) \quad \underline{BV}(MN) = \underline{BV}(M) \cup \underline{BV}(N)$$

For instance,

$$\begin{aligned} & \underline{BV}((\lambda x\lambda y. x)\lambda ab) \\ &= \underline{BV}(\lambda x\lambda y. x) \cup \underline{BV}(\lambda ab) && (BV3) \\ &= \underline{BV}(\lambda x\lambda y. x) \cup \{a\} \cup \underline{BV}(b) && (BV2) \\ &= \underline{BV}(\lambda x\lambda y. x) \cup \{a\} \cup \emptyset && (BV1) \\ &= \{x\} \cup \underline{BV}(\lambda y. x) \cup \{a\} && (BV2) \\ &= \{x\} \cup \{y\} \cup \underline{BV}(x) \cup \{a\} && (BV2) \\ &= \{x\} \cup \{y\} \cup \emptyset \cup \{a\} && (BV1) \\ &= \{x, y, a\} \end{aligned}$$

Similarly, the set $\underline{FV}(M)$ of free variables of the term M is formed as follows:

$$(FV1) \quad \underline{FV}(x) = \{x\}$$

$$(FV2) \quad \underline{FV}(\lambda xM) = \underline{FV}(M) \setminus \{x\}$$

$$(FV3) \quad \underline{FV}(MN) = \underline{FV}(M) \cup \underline{FV}(N)$$

For example:

$$\begin{aligned}
 \underline{FV}(\lambda x. Fx) & \\
 &= \underline{FV}(Fx) \setminus \{x\} && \text{(FV2)} \\
 &= (\underline{FV}(F) \cup \underline{FV}(x)) \setminus \{x\} && \text{(FV3)} \\
 &= (\{F\} \cup \{x\}) \setminus \{x\} && 2 \times \text{(FV1)} \\
 &= \{F\}
 \end{aligned}$$

Programs computing the sets \underline{BV} and \underline{FV} can be found in the appendix (pg 149).

Substitution

Using the concept of free and bound variables, more precise rules for the *substitution* $M[x:=N]$ can now be defined (by [Church], but see [Hankin] for a more compact presentation):

- (S1) $x[x:=N] \equiv N$
- (S2) $y[x:=N] \equiv y$
- (S3) $(\lambda xM)[x:=N] \equiv \lambda xM$
- (S4) $(\lambda xM)[y:=N] \equiv \lambda x(M[y:=N]),$
if $x \notin \underline{FV}(N)$ or $y \notin \underline{FV}(M)$
- (S5) $(\lambda xM)[y:=N] \equiv \lambda z(M[x:=z])[y:=N],$
if $x \in \underline{FV}(N), y \in \underline{FV}(M), z$ not in λxM
- (S6) $(MN)[x:=P] \equiv (M[x:=P])(N[x:=P])$

The notation $M[x:=N]$ was introduced by Barendregt, while Church used the “substitution” operator $S_N^x M$. The notation $M[x/N]$ is also used, but less common.

Rule (S1) basically says that substituting a term N for a (free) variable x results in the term N .

Rule (S2) says that variables not subject to a specific substitution will not be affected by that substitution, e.g. substituting x will not change any occurrences of y .

Rule (S3) states that bound variables are unaffected by substitution, e.g. $(\lambda xx)[x:=N] \equiv \lambda xx$.

Rule (S4) defines naive substitution as described earlier (pg 14). This rule covers the case where no name capturing takes place during substitution. (Note: the $y \notin \underline{FV}(M)$ part covers the trivial case where no substitution would take place at all.)

Rule (S5) deals with name capturing. It basically says that when a name bound in a term P would be captured during substitution, that name will be replaced with a different name (z) first, where z does not appear in P (neither bound nor free). For example:

$(\lambda x \lambda y. xy)[x:=y]$ is converted to $(\lambda x \lambda z. xz)[x:=y]$

Rule (S6), finally, covers substitution in applications by inductively applying the rules (S1) through (S6) to both the function M and argument N in (MN) .

Conversion and Reduction

Conversion is the process of rewriting a term according to some specific rules. In this section, the rules of λ -conversion will be discussed, i.e. the rules for rewriting λ -terms.

Substitution rule (S6) contains the following conversion:

(Alpha) $\lambda x M \rightarrow_{\alpha} \lambda y (M[y:=x])$
if $y \notin \underline{FV}(M)$ and $y \notin \underline{BV}(M)$

This conversion is called α -conversion and it indicates that the names of bound variables can be substituted in λ -terms, as long as the new name is not already contained in the term. E.g.: λxx is the same as λyy or λaa . We say that these terms are “equal modulo α -conversion” and write:

$\lambda xx \equiv_a \lambda yy \equiv_a \lambda aa$

Note that

$\lambda xy \not\equiv_a \lambda yy$

because $y \in \underline{FV}(\lambda xy)$.

In the remainder of this text, syntactic equality will always be considered modulo α -conversion, so $M \equiv N$ will always denote $M \equiv_{\alpha} N$.

The application rule (App) is also known as β -conversion:

$$\text{(Beta)} \quad (\lambda xM)N \rightarrow_{\beta} M[x:=N]$$

where $M[x:=N]$ indicates substitution according to the rules (S1) through (S6).

Multiple β -conversion steps may be possible in the same term:

$$\begin{aligned} & (\lambda xyz. xz(yz)) (\lambda xy. x) (\lambda xy. x) \\ & \rightarrow_{\beta} (\lambda yz. (\lambda xy. x)z (yz)) (\lambda xy. x) \\ & \rightarrow_{\beta} \lambda z. (\lambda xy. x)z ((\lambda xy. x)z) \\ & \rightarrow_{\beta} \lambda z. (\lambda y. z) ((\lambda xy. x)z) \\ & \rightarrow_{\beta} \lambda z. (\lambda y. z) (\lambda y. z) \\ & \rightarrow_{\beta} \lambda zz \end{aligned}$$

In general, M is said to be β -reducible (or just *reducible*) to N , if there is a finite number of reduction steps that converts M into N . This is indicated by the notation $M \twoheadrightarrow_{\beta} N$. Formally:

$$\text{(BR)} \quad \frac{M \equiv N \text{ or } (M \rightarrow_{\beta} L \text{ and } L \twoheadrightarrow_{\beta} N)}{M \twoheadrightarrow_{\beta} N}$$

The above notation is borrowed from *sequent calculus* (see [Gentzen]). It is a formalization of logical reasoning, where the premises or propositions are listed above a horizontal bar and the conclusions under the bar.

When exactly one single β -reduction step is required to transform M into M , M is said to be *immediately reducible* to N . When zero or more steps are required, M is said to be *reducible* to N .

So above rule (BR) states, “if M is syntactically equal (modulo α -conversion) to N or immediately reducible to L and L is reducible to N , then M is reducible to N .”

Note that definition (BR) includes the trivial case $M \equiv N$, which requires zero reduction steps.

In the following, $M \twoheadrightarrow N$ will indicate $M \twoheadrightarrow_{\beta} N$ and $M \rightarrow N$ will denote $M \rightarrow_{\beta} N$, unless stated otherwise.

Compilation

The Scheme language (see [R4RS]) can be considered to be an implementation of λ -calculus, although there are some significant differences, as will be shown in the next chapter (pg 121).

In this chapter the interpretation and compilation of a subset of Scheme will be discussed. First, the classic model of LISP and Scheme interpretation will be introduced. Then an alternative model of interpretation will be explained and the focus will shift from λ -calculus towards Scheme. The chapter will conclude with the translation of a small Scheme subset to portable C code.

The Language

The following is a formal summary of the Scheme subset to be used in this chapter. We will call this subset *Scheme*₀ and advance the index as more definitions will be added to the language. Of course, abstraction and application are the only parts of the language that would really be required, but we add some further definitions for convenience. Definitions can be thought of to be schematic definitions (pg 37):

(L1) **(lambda (v) e)** $\rightarrow (\lambda ve)$

(L2) **(e1 e2)** $\rightarrow (e_1 e_2)$

(L3) **(if e1 e2 e3)** $\rightarrow [\text{if } e_1; e_2; e_3] \rightarrow ((e_1(\lambda xe_2)(\lambda xe_3))(\lambda xx))$

(L1), (L2), and (L3) are just alternative syntaxes for λ -abstraction, application, and propositional functions. Note that **lambda** is really limited to a single variable for now. This restriction will be lifted later (pg 79).

In (L3), note that e_1 will be replaced with $F \equiv (\lambda xy. y)$, if $e_1 \equiv \text{nil}$ and with $T \equiv (\lambda xy. x)$ otherwise, thereby translating Scheme truth-values to λ -calculus.

(L4a) **(quote e)** $\rightarrow \lambda xe$

(L4b) **(quote e)** $\rightarrow [\text{nf } e]$

Quotation is a concept that cannot easily be translated to λ -calculus. It is used in Scheme to refer to *data objects*, a concept not known in λ -calculus. A data object may be thought of as a term that is not to be evaluated.

Hence there are two approaches to representing quoted objects. One is to use λ -abstraction to create a term in *head normal form* (L4a), the other would be to create a special schematic definition which denotes that a term already *is* in normal form (L4b).

Both approaches have their advantages and drawbacks. (L4a) maps **quote** to pure λ -calculus, but in order to access the quoted value, the resulting abstraction has to be applied to a value, which is not how quoted objects are defined in Scheme.

(L4b) avoids this problem, but at the cost of adding terms to λ -calculus that have the shapes of redexes but the semantics of normal forms.

Due to these issues quotation will be handled rather informally in the following.

The remaining definitions are merely syntactic sugar for the functions dealing with *kons* pairs and lists:

$$(L5) \quad (\mathbf{cons} \ e_1 \ e_2) \rightarrow (\underline{kons} \ e_1 \ e_2) \rightarrow ((\lambda xykn. kxy)e_1 e_2)$$

$$(L6) \quad (\mathbf{car} \ e) \rightarrow (\underline{kar} \ e) \rightarrow ((\lambda k. (k \lambda xy. x) \lambda x. \underline{nil})e)$$

$$(L7) \quad (\mathbf{cdr} \ e) \rightarrow (\underline{kdr} \ e) \rightarrow ((\lambda k. (k \lambda xy. y) \lambda x. \underline{nil})e)$$

$$(L8) \quad (\mathbf{pair?} \ e) \rightarrow (\underline{konsp} \ e) \rightarrow ((\lambda k. (k \lambda xy. T) \lambda xF)e)$$

$$(L9) \quad (\mathbf{null?} \ e) \rightarrow (\underline{nullp} \ e) \rightarrow ((\lambda k. (k \lambda xy. F) \lambda xT)e)$$

Note that the propositional functions **pair?** and **null?** return **()** for falsity and **(quote t)** for truth. The **if** syntax follows [R4RS] and allows the empty list **()** to represent falsity. The more modern **#t** and **#f** forms are absent.

Abstract Reduction Machine

Languages based on λ -calculus are typically interpreted by *abstract machines*, i.e. computer programs that implement a set of rules for interpreting formal languages.

A special case of the abstract machine is the *metacircular interpreter*, where the language being interpreted (source language, S) and the implementation language (I) of the interpreter are the same, so that primitive functions of S can directly be implemented using the corresponding functions of I. See [McCarthy] for an example.

In this chapter the principle of metacircular interpretation will be explored and combined with the *translation* of programs to a more suitable form for interpretation by actual computer hardware.

A Naive Compiler

Here is a simple *compiler* that translates *Scheme*₀ to a language suitable for abstract interpretation. The following transformations are performed:

Source	Target	Note
v	<i>v</i>	variable
(quote e)	<i>(%quote e)</i>	quotation
(lambda (v) e)	<i>(%lambda (v) e)</i>	abstraction
(if p c a)	<i>(%if p c a)</i>	selection
(e1 e2)	<i>(%apply e₁ e₂)</i>	application
	<i>(%tail-apply e₁ e₂)</i>	application

Most terms are kept as they are, but a *%*-prefix is added to mark keywords as machine instructions (which means that the “%” character must be excluded from the alphabet of the source language in order to avoid the accidental injection of instructions).

Only application is transformed into either an *%apply* or *%tail-apply* instruction. The type of the application instruction depends on the context in which the application appears. In each line, **a** indicates an application and **t** indicates a *tail application*:

- (T1) **(if a t t)**
- (T2) **(lambda (v) t)**
- (T3) **(t (a ...))**

In every other context, application is an ordinary (non-tail) application. Note in particular: **(t (a e))**, i.e. only the outermost application in a composition of functions is in a tail position.

The meaning of tail application will be explained in a following section, dealing with abstract interpretation (pg 55).

Here is the Scheme code to the compiler performing the above transformations:

```
(define (comp x)

  (define (comp-expr x t)
    (cond ((not (pair? x))
           x)
          ((eq? 'quote (car x))
           `(%quote ,(cadr x)))
          ((eq? 'lambda (car x))
           `(%lambda ,(cadr x)
              ,(comp-expr (caddr x) #t)))
          ((eq? 'if (car x))
           `(%if ,(comp-expr (cadr x) #f)
                 ,(comp-expr (caddr x) t)
                 ,(comp-expr (caddrr x) t)))
          (else
           `((if t '%tail-apply '%apply)
              ,@(map (lambda (x)
                       (comp-expr x #f))
                     x))))))

  (comp-expr x #t))
```

Bindings and Free Variables

In λ -calculus, application is implemented by substitution (see rules (S1) through (S6), pg 17). However, substitution is impractical for interpretation by a computer, because it potentially has to replace values for variables in a lot of different places in a term. E.g., in

$(\lambda x. xxx)M$

three instances of the variable x would have to be replaced during beta reduction (rule (Beta), pg 19).

Hence an extension to λ -calculus is introduced here that allows to keep track of substitutions without actually performing them:

$$(\lambda x. xxx)M \rightarrow [x = M]xxx$$

This is an alternative syntax for the substitution

$$xxx[x := M]$$

but we will in particular use the new notation to keep track of substitutions performed in abstractions, e.g.:

$$(\lambda xy. x)M \rightarrow \lambda y. [x = M]x$$

In general:

$$\begin{aligned} \text{(Env1)} \quad & (\lambda x_1 \cdots x_n. M)N_1 \cdots N_n \\ & \equiv (\lambda x_1 \cdots x_n. []M)N_1 \cdots N_n \\ & \rightarrow (\lambda x_2. [x_1 = N_1]M)N_2 \cdots N_n \\ & \twoheadrightarrow [x_n = N_n; \cdots; x_1 = N_1]M \end{aligned}$$

$$\begin{aligned} \text{(Env2)} \quad & [x_n = N_n; \cdots; x_1 = N_1]M \\ & \equiv M[x_n = N_n] \cdots [x_1 = N_1] \\ & \text{if } x_1, \cdots, x_n \notin \bigcup_{i=1}^n \underline{FV}(N_i) \end{aligned}$$

The $[x = M]$ part in $\lambda y. [x = M]N$ is called the *environment* of the term. An environment consists of a list of *variable-term* pairs $x = M$, where each pair indicates that the variable x is *bound to* the value M . Each variable in the term M that appears in the environment is to be replaced with the value to which it is bound. An environment may be empty, which is denoted by $[]$.

This approach is equivalent to multiple (i.e. zero or more) substitutions as shown in (Env2), as long as no variable in the environment occurs free in a term of the environment. We will later see that this precondition is trivially fulfilled in the implementation (pg 60).

Note particularly that, due to (Env1),

$$\begin{aligned} (\lambda xx. M)NP &\rightarrow (\lambda x. [x = N]M)P \\ &\rightarrow [x = P; x = N]M \\ &\equiv M[x = P] \end{aligned}$$

because after the substitution $M[x = P]$, there are no free instances of x left in M . Technically speaking, variables of inner scopes take precedence over variables of outer scopes. This phenomenon is known as *shadowing* of variables in programming jargon. It is consistent with the reduction

$$(\lambda xx. x)NP \twoheadrightarrow P$$

Abstract Interpretation

The code presented in this section *evaluates* expressions compiled by the compiler discussed previously. In this context, “evaluating an expression” basically means to reduce it to its normal form mechanically. All evaluators described in the following reduce *restricted* $\lambda K\beta$ -calculus and do not reduce head normal forms.

The abstract machine performing the reduction operates on four objects:

- the expression x
- the environment E
- the stack S
- the continuation C

The *expression* x is a compiled $Scheme_0$ program as described in the previous sections.

The *environment* E is an ordered set of variable bindings as described in the previous section, like the $[x = N]$ part in the term $\lambda y. [x = N]M$. Note that E is merely a global pointer to the (possibly local) environment currently in effect. More on that below.

List of Figures

1	Abbreviations	15
2	Diamond property	23
3	Logic operators	32
4	List structure	40
5	Environments	54
6	Environment propagation	65
7	Boxing	75
8	Call frame, upward-growing stack	90
9	Collecting variadic arguments	93
10	Tail call optimization	95
11	Tail call optimization with constant frame size	135
12	Environment pruning	138

List of Definitions

(W1)–(W3)	Well-formed formulae	14
(App)	Application	14
(A1)–(A5)	Abbreviations	15
(BV1)–(BV3)	Bound variables	16
(FV1)–(FV3)	Free variables	16
(S1)–(S6)	Substitution	17
(Alpha)	α -conversion	18
(Beta)	β -conversion	19
(BR)	β -reduction	19
(Conv)	(β -)convertibility	20
(Rest)	Restricted λK -calculus	22
(Conf)	Confluence	23
(Ext)	Extensionality	24
(η)	η rule	24
(S4a)	Substitution with variable convention	25
(Ren)	Renaming of bound variables	26
(FP)	Fixed Points	33
(List)	Lists	39
(List2)	Lists with end of list	39
(L1)–(L9)	<i>Scheme</i> ₀	45
(T1)–(T3)	<i>Scheme</i> ₀ tail positions	47
(Env1),(Env2)	Environments	49
(EncT)	Enclosing terms	58
(EncA)	Enclosing abstractions	58
(LCtx)	Lexical context	58
(Closed)	Closed terms	58
(Env1v),(Env2v)	Environment vectors	60
(L10)	<i>Scheme</i> ₁ , begin	71
(L11)	<i>Scheme</i> ₁ , set!	72
(T4)	<i>Scheme</i> ₁ add. tail positions	72
(L1b)	<i>Scheme</i> ₁ implicit sequences	72
(Appm)	Application with multiple variables	79
(SL)	Substitution lemma	79
(Err1),(Err2)	Errors	85

(AR)	Argument restriction	121
(A5b)	Implementation of (A5)	121
(A3b)	Implementation of (A3)	121
(LL1)–(LL6)	LC-based language definition	122
(AA1)–(AA3)	LC-based language abbreviations	122
(TT1)–(TT4)	LC-based language tail positions	127

Index

- α -conversion 18, 25
- β -convertibility 20
- β -expansion 125
- β -normal form 20
- β -redex 20
- β -reduction 19
- η -reduction 29
- λ -calculus 13
- λ -conversion 18
- λ -lifting 125
- $\lambda\beta$ -calculus 22
- $\lambda\eta$ -calculus 24
- λK -calculus 14
 - restricted 22, 36 50
- abbreviation 15
- abstract machine 46
- abstraction 14
 - enclosing 58
- address store 91
- alphabet 13
- alternative 37
- application 14, 48, 52
 - of non-function 86
- applicative order
 - reduction 21, 35
- argument 14
- associativity 15
- binary function 123
- binding 15, 51, 51
- boxing 74, 138
- branching 106
- call frame 90
- callee 65, 95
- caller 65, 95
- capturing 15
- Church numeral 27
- Church pair 38
- closure 61
 - conversion 61, 126
- combinator 137
- compiler 47
- complexity 57
- composition 28
- confluence 23
- conj function 32
- conjunction 32
- cons cell 41
- consequent 37
- constancy function 22
- context 51
 - global 144
- continuation 51
- conversion 18
- convertibility 20
- current environment 52
- currying 32
- data object 46
- definition 15
 - schematic 37
 - syntactical 15
- disj function 32
- disjunction 32
- div function 38
- division 34
- effect 52, 71
 - ordering 71

- enclosing abstraction 58
- enclosing term 58
- environment 49, 50, 91
 - current 52
 - initial 51
- environment propagation 65
- environment pruning 138
- environment vector 60
- eqp function 33
- equality
 - extensional 24
 - intensional 24
- equivalence 15, 18
- equivalence class 25
- evaluation 21, 50
 - of arguments 82
- exp function 29
- expression 20, 50
- extensionality 24
- fixed point 33
- fixed point combinator 34
- formal system 13
- free variable 16
- function 14
 - binary 123
 - nullary 79
 - primitive 53, 67, 67
 - propositional 31, 37
 - unary 123
 - variadic 83
- function definitions
 - conj 32
 - disj 32
 - div 38
 - eqp 33
 - exp 29
 - geqp 33
 - gtp 33
 - kar 41
 - kdr 41
 - kons 41
 - konsp 41
 - leqp 33
 - ltp 33
 - minus 31
 - neg 32
 - neqp 33
 - nullp 41
 - P (predecessor) 30
 - plus 28
 - plus 31
 - S (successor) 27
 - times 28
 - zerop 32
- function call 56
- function object 52
- function term 98
- geqp function 33
- global context 144
- gtp function 33
- head normal form 24, 46, 50
- hole 51
- initial environment 51
- inlining 67
- innermost 21
- instruction store 91
- jump-around-jump 99
- kar function 41
- kdr function 41
- kons function 41
- konsp function 41
- leqp function 33

- lexical context 52, 58
- list 39
- literal 91
- literal pool 91
- location 74
- loop unrolling 96
- ltp function 33
- macro 143
- macro expansion 143
- metacircular
 - interpretation 47, 143
- minus function 31
- modified subtraction 29
- mutation 71
- neg function 32
- negation 32
- neqp function 33
- normal form 20
 - head 24, 46, 50
 - principal 26, 59
- normal order reduction 21, 35
- nullary function 79
- nullp function 41
- ordered pair 38
- P (predecessor) function 30
- pair, Church-encoded 38
- plus function 28, 31
- predecessor 29
- primitive function 53, 67, 67
- principal normal form 26, 59
- program 42
- propositional function 31, 37
- quasi-quotation 143
- quotation 46
- reader 112
- recursion 34, 35
- redex 20
- reduction 19
 - applicative order 21, 35
 - normal order 21, 35
- register machine 102
- relational operator 33
- renaming 26, 138
- restricted
 - λK -calculus 22, 36, 50
- S (successor) function 27
- save (S9 Core) 110
- schema variable 142
- schematic definition 37
- schönfinkelization 32
- semantics 67
- sequencing 71
- sequent calculus 19
- shadowing 50
- stack 51, 91
- stack machine 89
- substitution 14, 17, 25, 141
- substitution lemma 79
- sub-term 15
- successor 27
- syntactic sugaring 37
- tail application 47
- tail call conversion 127
- tail call elimination 56, 95
- tail call optimization 56, 95
- tail position 55
- tail recursive 139
- term 49, 98
 - enclosing 58
- times function 28
- truth value 31

- unary function 123
- unboxing 75
- unsave (S9 Core) 110
- variable 14, 27, 49, 71
 - bound 15
 - free 16
- variable look-up 57
- variadic function 83
- virtual machine 89
- VM 89
- well-formed formula 13, 85
- wff 13, 85
- Y combinator 34
- zerop function 32