Journal of Universal Computer Science, vol. 23, no. 1 (2017), 112-126
submitted: 16/9/16, accepted: 8/12/16, appeared: 28/1/17 © *J.UCS*

First-Class Functions in an Imperative World

Roberto Ierusalimschy
(PUC-Rio, Brazil
roberto@inf.puc-rio.br)

Abstract: First-class functions are a hallmark of functional languages, but they are a useful concept in imperative languages, too. Even ANSI C offers a restricted form of first-class functions (in the form of pointers to functions), and several more recent imperative languages, such as Python 3, Go, Lua, and Swift, offer first-class, anonymous functions with lexical scoping.

In most imperative languages, however, first-class functions are an advanced feature used by seasoned programmers. Lua, by contrast, uses first-class functions as a building block of the language. Lua programmers regularly benefit from diverse properties of its functions for routine constructions such as exception handling, module definitions, object-oriented programming, and iterators. Moreover, first-class functions play a central role in the API between Lua and C.

In this paper, we present some aspects of Lua that were designed with first-class functions in mind, in particular its module system, exception handling, iterators, facilities for object-oriented programming, and the API between C and Lua. We also discuss how those different aspects of Lua use different properties of first-class functions to achieve two important goals in the design of Lua, namely small size and embeddability (its easiness of interfacing with other languages).

Key Words: Lua, scripting languages, functional languages
Category: D.3.3, D.1.1

1 Introduction

A key feature of a functional language is the presence of functions as first-class values and anonymous functions with lexical scoping. First-class functions are a hallmark of functional languages, but they are a useful concept in imperative languages, too.[1] Many imperative functions may be more generic if coded as higher-order functions. For instance, even in C the standard sort function (`qsort`) is a higher-order function, with its comparison function given as an argument in the form of a function pointer. Callback functions are another example of the usefulness of first-class functions in imperative languages. Moreover, imperative languages that support first-class functions allow programmers to borrow useful programming techniques from the functional world. Nonetheless, even in imperative languages with proper support for first-class functions, such as Go, Swift, or Javascript, you can work reasonably well without ever resorting to features like anonymous functions or lexical scoping.

[1] Here I am using the term “first-class function” loosely. Later I will define more precise terms.

Lua [Ierusalimschy et al., 1996, Ierusalimschy, 2016] is a imperative scripting language widely used in the game industry, in embedded systems, and as a configuration language in general[2]. (This paper was formatted with LuaTEX, the successor of pdfTEX that uses Lua as an embedded scripting language.) Lua is known for its tables [Sebesta, 2009], which are associative arrays where both keys and values can have any type; we seldom emphasize the role of first-class functions in the language. However, Lua uses first-class functions extensively. Its standard library contains several higher-order functions: `sort` takes a comparison function; `gsub`, which does pattern matching and replacement on strings, can receive a *replacement function* that receives the original text matching the pattern and returns its replacement.

However, in Lua, more than a powerful mechanism for programmers, first-class functions are an *empowering* mechanism for the language itself. Several other mechanisms in the language are built on top of first-class functions. In particular, several important applications of tables are based on their ability to have functions as fields (e.g., modules and classes). Moreover, first-class functions are a key ingredient in keeping the language small and embeddable.

In this paper, we describe how Lua supports first-class functions and discuss how they allow and empower other Lua features, including modules, object-oriented facilities, exception handling, and the Lua-C API. Section 2 classifies the main properties of first-class functions in programming languages and discusses how Lua supports these properties. Section 3 shows how different mechanisms in Lua benefit from different properties of first-class functions. Section 4 discusses the use of first-class functions in the Lua-C API, which is a particularly important aspect of Lua. Finally, Section 5 draws some conclusions.

2 First-Class Functions

First-class functions, also known as *closures*, *anonymous functions*, or simply *lambdas*, means different things to different people. So, first, we will try to distill its main properties. Functions in functional languages have four important properties:

- Functions are *first-class values.* They are values with the same rights of other values: they can be passed as parameters, returned as results, be part of structured data, etc.
- Functions can be nested, that is, we can define functions inside other functions.

[2] `https://en.wikipedia.org/wiki/Category:Lua-scripted_video_games`
`https://en.wikipedia.org/wiki/Lua_(programming_language)#Applications`

– There is some syntax for *anonymous functions* (also called function literals). We can create a function inside any expression, right in the place where we need it.

– Functions respect *lexical scoping*. Nested functions have full access to variables defined in enclosing functions, even when the function *escapes* its enclosing function (that it, it remains accessible after the end of the enclosing invocation).

All these properties were present in the lambda calculus, and they are present in virtually any functional language. Nevertheless, and despite the close interaction among them, these properties are somewhat independent. For instance, C offers functions as first-class values (in the form of pointer to functions), but does not have nested functions and, therefore, no anonymous functions nor lexical scoping [ISO C, 2000]. The original Lisp had first-class anonymous functions, but lacked lexical scoping [McCarthy, 1960]. Pascal has nesting and lexical scoping, but does not have syntax for anonymous functions and function values are not first class. (They can be passed as arguments, but cannot be returned nor assigned, to avoid escapes.) Modula-2 has both functions as first-class values and lexical scoping, but not together: only non-nested procedures are first-class values [Wirth, 1985]. Python 3 have both functions as first-class values and lexical scoping, but only a restricted form of anonymous functions. (The body of a lambda expression must be a single expression; it cannot be generic Python code [Python, 2015].)

Imperative languages have one more important property connected to lexical scoping. When a function is created in a functional language, it can capture either the values of its external variables or the variables themselves; since all variables are immutable, the resulting semantics is the same. For imperative languages, the semantics change. A few imperative languages capture values. (An example is Java, which disguises the problem by restricting that local variables used in lambda expressions must be final/immutable [Gosling et al., 2015].) Several others, following the lead from Scheme, capture the variables themselves (e.g., Lua, Swift, and Go), so that an update made by a function is seen by all functions sharing that variable.

When the language captures variables, it is easy to capture values, as the next Lua fragment illustrates:

```
-- assume 'x' is a variable visible here
do
  -- make an immutable private copy of 'x',
  -- only visible inside this block
  local x = x
```

```
    -- the next function will form a closure with
    -- this immutable copy
    function foo (y) return x + y end
  end
```

However, when the language captures values, there is no direct way to capture variables. The option is to box the variable inside a structure and share its immutable reference, as we do with mutable data in Standard ML. For a mostly-functional language like Standard ML, these occasional boxings seem unproblematic, but for a mostly-imperative language like Lua, they can be quite cumbersome.

Moreover, when a language captures variables, we can enclose any piece of code inside an anonymous function with little change. (We only have to worry with statements that invoke escape continuations, such as `break` and `return`.) When the language captures only values, any assignment inside the block being enclosed becomes a problem.

Lua went through several stages until it reached its current support for functions [Ierusalimschy et al., 2007]. Since its first version, released in 1993, functions in Lua have been first-class values. However, these older versions did not support anonymous functions (function literals) nor nested functions, and therefore static scoping was not an issue. In 1998, Lua 3.1 introduced support for anonymous functions. The usual syntax for defining a function in Lua is like this:

```
function add (x, y)
  return x + y
end
```

Since version 3.1, this syntax is only sugar for the next fragment:

```
add = function (x, y)
  return x + y
end
```

That is, the code creates an anonymous function and assigns it to the global variable `add`. Therefore, all functions in Lua are anonymous, like in Scheme. A "name" for a function is actually the name for a variable that holds that function.

However, Lua 3.1 still did not support proper lexical scoping (with access to variables, not their values). Instead, Lua introduced the concept of *upvalues*, which allowed nested functions to access the values of external variables, but not the variables themselves. An upvalue represents the value of an external variable frozen when the closure is created, as illustrated in the next example:

```
local x = 10
function foo () return %x end
x = 20
print(foo())            --> 10
```

In these older versions of Lua, an upvalue was accessed with an explicit operator, a prefixed percent sign (as the `%x` in the example). Its purpose was to remind the programmer that the code is not accessing the variable itself, but a (possibly outdated) copy of its value.

For several patterns (e.g., pure functions), upvalues are all we need. However, in an imperative language like Lua, where assignment is the norm, upvalues can become cumbersome. In 2003, Lua 5.0 introduced full lexical scoping, using a novel implementation technique [Ierusalimschy et al., 2005]. (That implementation is also safe for space.)

Currently, any lambda expression can be directly translated to Lua[3], which evaluates the expression using a strict semantics ("call-by-value"). As an example, consider the next definition for a factorial function, using the Z fixed-point combinator:

$$
\begin{aligned}
&\textbf{let}\ \ \mathrm{Z} = \lambda f.(\lambda x.f(\lambda v.((x\ x)v)))(\lambda x.f(\lambda v.((x\ x)v))) \\
&\qquad \mathrm{F} = \lambda f.\lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times f(n-1)\ \ \textbf{in} \\
&\qquad\ \ (Z\ F)\ 5
\end{aligned}
$$

Listing 1 shows how we can write the same code in Lua. Despite the verbosity, the Lua code is a direct translation of the lambda-calculus definition.

3 Functions in Lua

Many mechanisms in Lua take advantage of first-class functions, as outlined before. More than a powerful mechanism, first-class functions are an empowering mechanism in Lua. Several other mechanisms become better (or even possible) with the presence of first-class functions in the language. In this section we will cover some of these mechanisms, emphasizing how they benefit from each specific property supported by functions.

3.1 Eval

Since Lisp, most dynamic languages present an `eval` function, which allows execution of dynamically-created code inside the environment of the calling program. In fact, we may consider `eval` as a defining feature of a dynamic language.

Of course, any Turing-complete language can execute dynamically-created code, but not always in the same environment of the original program. As a simple example, consider the next fragment:

[3] Given that Lua is dynamically typed, it can fully represent the untyped lambda calculus.

```
local Z = function (f)
  return
    (function (x) return f (function (v) return x(x)(v) end) end)
    (function (x) return f (function (v) return x(x)(v) end) end)
end

-- an example: the factorial function
local F = function (f)
            return function (n)
                     if n == 0 then return 1
                     else return n * f(n - 1)
                     end
                   end
          end

print(Z(F)(5))    --> 120
```

Listing 1: The Z Fixed-Point Combinator in Lua

```
x = 1
-- 's' contains the string "y = x + 10"
eval(s)
print(y)        --> 11
```

Note how the code in `s` both accessed and defined entities in the environment of the calling program.

Being a dynamic language, Lua also offered `eval` functions in its early versions, called `dostring` (to evaluate the contents of a string) and `dofile` (to evaluate the contents of a file). However, later we changed that to a "compile" primitive, called `load`. The function `load` is a higher-order function, which receives a text (or a function that returns the text piecemeal) and returns an anonymous function equivalent to the given arbitrary text. Of course, *eval* and *load* are equivalent—given one of them, it is easy to implement the other:

```
function eval (code)
   -- compiles source 'code' and executes the result
   return load(code)()
 end
```

```
function load (code)
  -- creates an anonymous function with the given body
  return eval("return function () " .. code .. " end")
end
```

(The infix two-dot operator '`..`' denotes string concatenation in Lua.) However, we consider that `load` is a somewhat better primitive, because it clearly separates the transformation from data into code embedded in the running program (which refines the reflective essence of a dynamic language) from the actual execution of the resulting code (which is a regular function call). In particular, unlike `eval`, `load` is a pure and total function.

An important property of *load* is that it compiles any chunk of code as the body of an anonymous function, with all functions defined by the chunk nested inside it. Therefore, a chunk can declare local variables, which are equivalent to `static` variables in C: they are visible only inside the chunk, they are visible to all functions inside the chunk, and they preserve their values between successive calls to these functions.

3.2 Modules

Modules in Lua are just tables (associative arrays) populated with functions. When we write `math.log(x)`, Lua sees the code as `math["log"](x)`, that is, a call to the function at index `"log"` from the table at variable `math`.

Because both modules themselves and their functions are first-class values in the language, several facilities come for free. For instance, it is trivial to rename modules and functions, by assigning them to local variables:

```
local m = require "math"
local s = m.sin
print(s(0), m.cos(0))   --> 0.0    1.0
```

For the implementation of modules, the key property of functions is that they are first-class values. However, lexical scoping is very handy, as they allow the declaration of private functions and values (not exported by the module). We simply define them as local variables in the main function creating the module.

3.3 Exception handling

Lua offers exception handling through two functions, `pcall` (Protected call) and `error`. The function `pcall` receives a function as an argument and runs that function in protected mode, so that any error that occurs during the execution of that function is contained by pcall. In languages like Java or JavaScript, exception handling goes like this:

```
try {
  <block/throw>
}

catch (err) {
  <exception code>
}
```

In Lua, we could write an equivalent code like this:

```
local ok, err = pcall(function ()
  <block/error>
end)

if not ok then
  <exception code>
end
```

Note the idiomatic indentation: it plays down the role of the anonymous function, emphasizing instead the whole construction, with pcall and the block of code. The function error throws an error, finishing the last active pcall invocation. It has a single argument, an *error value*. In case of errors, pcall returns false (to signal the error) plus this error value as a second result. Therefore, in the last fragment the exception code can access this error value through the variable err.

The function pcall per se only needs first-class values, as it is a higher-order function.[4] However, as the previous code fragment illustrates, pcall is much more convenient when the language supports anonymous functions with lexical scoping, because we can write the code being protected just like regular code. Not by chance, pcall appeared in Lua in the same release that brought anonymous functions.

3.4 Iterators

Several languages nowadays, such as Java and Swift, offer iterators based on OO facilities. In Lua, they are based on impure first-class functions.

[4] In fact, the argument to pcall never escapes. So, strictly, it does not need first-class values: restricted function values, like in Pascal, would be enough here.

The use of iterators is straightforward. For instance, the following fragment prints all lines of a file:

```
for l in io.lines(filename) do
  print(l)
end
```

Similarly, the next one prints all words from a string `s`:

```
for w in string.gmatch(s, %w+) do
  print(w)
end
```

(The pattern '`%w+`' describes a word as one or more alphanumeric characters.)

In Lua, an iterator is an (impure) function that, each time it is called, returns a "next" element of the iteration (or nil, to signal the end of the iteration). What we write in the `for` construction is usually a call to a factory function that returns the iterator function. As a simple example, the next factory creates iterators for numeric ranges:

```
function fromTo (n, m)
  return function ()
           if n > m then
             return nil    -- signal end of iteration
           else
             n = n + 1
             return n - 1
           end
         end
end
```

The function `fromTo` is the factory function. When called, it returns an anonymous function as the iterator. It can be used like here:

```
for x in fromTo(10, 12) do
  print(x)      -- will print 10, 11, and 12
end
```

Note that, as this example illustrates, lexical scoping is the key feature that allows the iterator to keep its state between steps. The example also shows how important it is for an iterator to update external variables (instead of only accessing their values), so that each call can advance the iteration. Note also that an iterator function always escapes its original scope (the factory function).

Again not by chance, the `for` construction was introduced in the same Lua version that introduced full lexical scoping (version 5.0).

3.5 Object-oriented programming

Lua adopts a "Do-It-Yourself" style of object-oriented (OO) programming. The language does not offer OO features directly; instead, it tries to offer mechanisms that allow us to implement the OO features that we need. Clearly, first-class functions is a key ingredient toward that goal.

In Lua, an object is a table (what else?): fields containing functions represent the object's methods; other fields represent its instance variables.[5] To allow methods to access these instance variables, we need a `self` (or `this`) argument. Lua provides this argument through a simple syntactic sugar. We can declare a method using the *colon notation*, as the next fragment illustrates:

```
function obj:add (p)
  self.x = self.x + p
end
```

Lua translates this fragment into the following code:

```
obj.add = function (self, p)
  self.x = self.x + p
end
```

That is, a method is a regular function whose first parameter is *self*. Similarly, there is a colon notation for calling methods. When we write `obj:add(12)`, Lua translates it to `obj.add(obj, 12)`, thus passing the receiver as the first parameter to the method.

Here, the main property from functions is being first-class values. Nesting, anonymous functions, and lexical scoping are of little direct use. Nevertheless, lexical scoping is useful for other OO mechanisms, like class attributes.

The separation between method selection (usually a regular table operation) and method invocation (a regular function call) also allows other useful features, such as the ability to call overridden methods, such as calls to `super`. It also simplifies the C API; because the API already has functions for table operations and for function calls, it can call Lua methods without any extra support.

4 The Lua-C API

A key feature of Lua is its API with C and other languages: the design of the entire language has taken the API into consideration [Ierusalimschy et al., 2011].

[5] An object in Lua does not have to actually contain all its methods as fields, because it can (and usually does) inherit fields from another table. If a table `A` inherits from `B`, and we compute `A`[k] where `k` is not present in `A`, the expression results in `B`[k]. Note that inheritance in Lua concerns only table accesses and it is orthogonal to what we are discussing here.

This API has been designed both for *extending* and for *embedding* Lua. Extending Lua means adding to Lua functions and types written in another language (typically C or C++), so that Lua code can use these functions and types as if they were native. Embedding Lua means allowing a program written in another language to call functions written in Lua for specific tasks. In both scenarios, first-class functions play an important role. Let us see first embedding, or how to call a Lua function from C.

All data exchange between Lua and C goes through a stack of Lua values. This stack has two purposes. The first one is to solve the mismatch between the dynamic typing in Lua and the static typing in C: all projections and injections between C values and Lua values go through the stack. The second purpose is to control garbage collection: C have direct access only to values in the stack, which cannot be collected while there.

As an example, consider the Lua fragment `r = f(x, y)`, where `x`, `y`, and `r` can have any type. How to code an equivalent operation in C? Assuming that all variables are global, the following code does the job:

```
lua_getglobal(L, "f");  /* push function */
lua_getglobal(L, "x");  /* push 1st argument */
lua_getglobal(L, "y");  /* push 2nd argument */
lua_call(L, 2, 1);  /* call function with 2 arguments */
lua_setglobal(L, "r");  /* pop the result into global 'r' */
```

(The first argument to all functions in the API, `L` in this example, is a reference to the Lua state being manipulated, which contains the stack.) Note how we retrieve the function `f` with the same function that we use for the other values.

As another example, to do the call `f(1, "x")` and get the result in a C integer variable, we can use the following code:

```
lua_getglobal(L, "f");  /* push function */
lua_pushinteger(L, 1);  /* push 1st argument */
lua_pushstring(L, "x");  /* push 2nd argument */
lua_call(L, 2, 1);  /* call function with 2 arguments */
int r = lua_tointeger(L, -1);
```

Injection functions (such as `lua_pushinteger` and `lua_pushstring`) always push the new Lua value on the top of the stack; projection functions (such as `lua_tointeger`) can access any value in the stack, using an index. (In the example, -1 means "first value from the top". A positive 1 would mean "first value from the bottom".)

Given the way functions are used in Lua, we can use `lua_call` to call anything. To call functions in modules, we only need to retrieve the function from the module table, using standard table operations. We can trivially call iterators inside loops in C code. To call methods, we retrieve the method (again using standard table operations) and add the receiver as a first argument in the call.

For error handling, we call the function with `lua_pcall`, which is the equivalent to pcall in the C API. First-class functions make this mechanism orthogonal to how we access the function to be called: we can use `lua_pcall` with functions in modules, methods, etc.

When extending Lua, the main point is how to to make functions written in C (or other languages) available to Lua scripts. Any C function must follow a certain protocol to work with Lua. First, it must have the following prototype:

```
typedef int (*lua_CFunction) (lua_State *L);
```

The sole parameter `L` is the state (and the stack) where the function should operate. When the function starts, its stack contains the call arguments. To return, the function pushes on the stack the values to be returned and returns (in C) the number of such values.[6] As an example, the following function implements the sine function for Lua:[7]

```
int sin_lua (lua_State *L) {
  double x = lua_tonumber(L, 1);
  lua_pushnumber(L, sin(x));
  return 1;
}
```

The call to `lua_tonumber` projects the first argument into a double, the call to `lua_pushnumber` pushes the sine of that double into the stack, and the return signals to Lua that the stack has one result.

The API offers one single primitive to inject C functions into a Lua value, `lua_pushcfunction`. It takes a pointer to the C function and creates a first-class value in Lua that represents that function. Everything else is done with regular value-manipulation primitives. For instance, to register our previous function `sin_lua` into the Lua global variable `sin`, we can use the following code:

```
lua_pushcfunction(L, sin_lua);
lua_setglobal(L, "sin");
```

As functions are first-class values, we can use the regular `lua_setglobal` to assign it into a global variable.

To create a module composed of C functions, we create a table and add to it the C functions that compose the module. Except for `lua_pushcfunction`, all we need are regular table-manipulation functions.

[6] Functions in Lua can return multiple values.
[7] The real implementation is more complex than that due to error handling.

```
int inner_add (lua_State *L) {
  double x = lua_tonumber(L, lua_upvalueindex(1));
  double y = lua_tonumber(L, 1);
  lua_pushnumber(L, x + y);
  return 1;
}

int add (lua_State *L) {
  double x = lua_tonumber(L, 1);
  lua_pushnumber(L, x);
  lua_pushcclosure(L, &inner_add, 1);
  return 1;
}
```

Listing 2: An example of a C closure

4.1 C closures

Of course, C functions cannot benefit from nesting or lexical scoping, but the Lua-C API offers an approximation, that we call *C closures*. When we register a C function into Lua, we can associate to it one or more arbitrary Lua values, which we call its *upvalues*. Whenever the C function is called from Lua, it can access its upvalues.

To illustrate the use of upvalues, consider the following curried *add* function:

```
function add (x)
  return function (y)
           return x + y
         end
end
```

Listing 2 shows how we can implement an equivalent function in C. In that code, the function add implements its homonymous function in the Lua code, while the function `inner_add` implements the nested anonymous function in the Lua code.

In the function add, the call to `lua_pushcclosure` injects the C function as a Lua function into the stack. This function is similar to `lua_pushcfunction`, but it associates a number of values (only one, in the example) on the top of the stack to the function being created, forming a *C closure*.[8] Whenever Lua calls a

[8] In fact, `lua_pushcfunction` is a macro that calls `lua_pushcclosure` with zero upvalues.

C closure, it makes those upvalues accessible to the C code.

In the function `inner_add`, `lua_upvalueindex` does the main trick. This macro retrieves from the Lua state the n-th upvalue (the first and only one, in the example) associated with the running function. The function then gets its first (and only) argument in the variable `y` and pushes (to return) the sum of its upvalue with its argument.

A C function can freely modify its own upvalues, but it cannot share those updates with other functions. When we need to share mutable data among several C functions, we have to box the data inside a table and set the table as an upvalue for all the functions involved.

5 Final Remarks

In 1960, Lisp [McCarthy, 1960] introduced first-class functions in programming languages. Few years later, the SECD machine used closures to implement lexical scoping efficiently [Landin, 1964]. Scheme brought lexical scoping into a Lisp-like language, showing to programmers the flexibility of the mix [Steele, 1976, Steele and Sussman, 1976, Steele, 1977]. David Turner paved the way to modern functional languages introducing case analysis by pattern matching, emphasizing lazy evaluation, and, more importantly, revolutionizing the evaluation of lazy languages with the use of combinators [Turner, 1975, Turner, 1979].

Imperative languages followed a more bumpy path. Only in the last few years first-class functions become more mainstream in those languages: Java, arguably the most used programming language nowadays, took almost 20 years to support anonymous functions, with Java 8 [Gosling et al., 2015]; even then, lambda functions in Java cannot access external mutable variables. Rust 1.0, released in 2015, also imposes restrictions on its first-class functions. ("Closures" in Rust either can access external mutable variables or can escape, not both [Rust, 2016].)

Several imperative languages now support first-class, anonymous functions with lexical scoping (e.g., Javascript, Swift, Python 3, Go). Nevertheless, to our knowledge, only Lua has first-class functions so ingrained in its design. Every code written in Lua is compiled as an anonymous function; modules, iterators, and objects in the language are based on first-class functions; exception handling is offered through a higher-order function. So, programmers gain from first-class functions in Lua from day one, even unknowingly.

References

[Gosling et al., 2015] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. (2015). *The Java Language Specification: Java SE 8 Edition.* Oracle.

[Ierusalimschy, 2016] Ierusalimschy, R. (2016). *Programming in Lua.* Lua.org, Rio de Janeiro, Brazil, fourth edition.

[Ierusalimschy et al., 1996] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (1996). Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652.

[Ierusalimschy et al., 2005] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2005). The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176. (SBLP 2005).

[Ierusalimschy et al., 2007] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2007). The evolution of Lua. In *Third ACM SIGPLAN Conference on History of Programming Languages*, pages 2.1–2.26, San Diego, CA.

[Ierusalimschy et al., 2011] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2011). Passing a language through the eye of a needle. *Communications of the ACM*, 54(7):38–43.

[ISO C, 2000] ISO C (2000). *International Standard: Programming languages C*. ISO. ISO/IEC 9899:1999(E).

[Landin, 1964] Landin, P. B. (1964). The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195.

[Python, 2015] Python (2015). *The Python Language Reference*. The Python Software Foundation, 3.5 edition.

[Rust, 2016] Rust (2016). The Rust programming language. `https://www.rust-lang.org/en-US/documentation.html`.

[Sebesta, 2009] Sebesta, R. (2009). Interview: Roberto Ierusalimschy — Lua. In *Concepts of Programming Languages*, pages 280–281. Addison-Wesley, ninth edition.

[Steele, 1976] Steele, G. L. (1976). Lambda: The ultimate declarative. Technical Report AI Memo No. 379, MIT, Cambridge, MA, USA.

[Steele, 1977] Steele, G. L. (1977). Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or LAMBDA, the ultimate GOTO. Technical Report AI Memo No. 443, MIT, Cambridge, MA, USA.

[Steele and Sussman, 1976] Steele, G. L. and Sussman, G. J. (1976). Lambda: The ultimate imperative. Technical Report AI Memo No. 353, MIT, Cambridge, MA, USA.

[Turner, 1975] Turner, D. A. (1975). SASL language manual. Technical Report CS/75/1, St. Andrews University.

[Turner, 1979] Turner, D. A. (1979). A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49.

[Wirth, 1985] Wirth, N. (1985). *Programming in Modula-2*. Springer-Verlag, third edition.