DOI:10.1145/3186277

Simplicity, small size, portability, and embeddability set Lua apart from other scripting languages.

BY ROBERTO IERUSALIMSCHY, LUIZ HENRIQUE DE FIGUEIREDO, AND WALDEMAR CELES

A Look at the Design of Lua

LUA IS A scripting language developed at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) that has come to be the leading scripting language for video games worldwide.[3,7] It is also used extensively in embedded devices like set-top boxes and TVs and in other applications like Adobe Photoshop Lightroom and Wikipedia.[14] Its first version was released in 1993. The current version, Lua 5.3, was released in 2015.

Though mainly a procedural language, Lua lends itself to several other paradigms, including object-oriented programming, functional programming, and data-driven programming.[5] It also offers good support for data description, in the style of JavaScript and JSON. Data description was indeed one of our main motivations for creating Lua, some years before the appearance of XML and JavaScript.

Our motto in the design of Lua has always been "mechanisms instead of policies." By policy, we mean a methodical way of using existing mechanisms to build a new abstraction. Encapsulation in the C language provides a good example of a policy. The ISO C specification offers no mechanism for modules or interfaces.[9] Nevertheless, C programmers leverage existing mechanisms (such as file inclusion and external declarations) to achieve those abstractions. On top of such basic mechanisms provided by the C language, policy adds several rules (such as "all global functions should have a prototype in a header file" and "header files should not define objects, only declare them"). Many programmers do not know these rules (and the policy as a whole) are not part of the C language.

Accordingly, in the design of Lua, we have replaced addition of many different features by creating instead only a few mechanisms that allow programmers to implement such features themselves.[6] The motto leads to a design that is economical in concepts. Lua offers exactly one general mechanism for each major aspect of programming: tables for data; functions for abstraction; and coroutines for control. On top of these building blocks, programmers implement several other features, including modules, objects, and environments, with the aid of minimal additions (such as syntactic sugar) to the language. Here, we look at how this motto has worked out in the design of Lua.

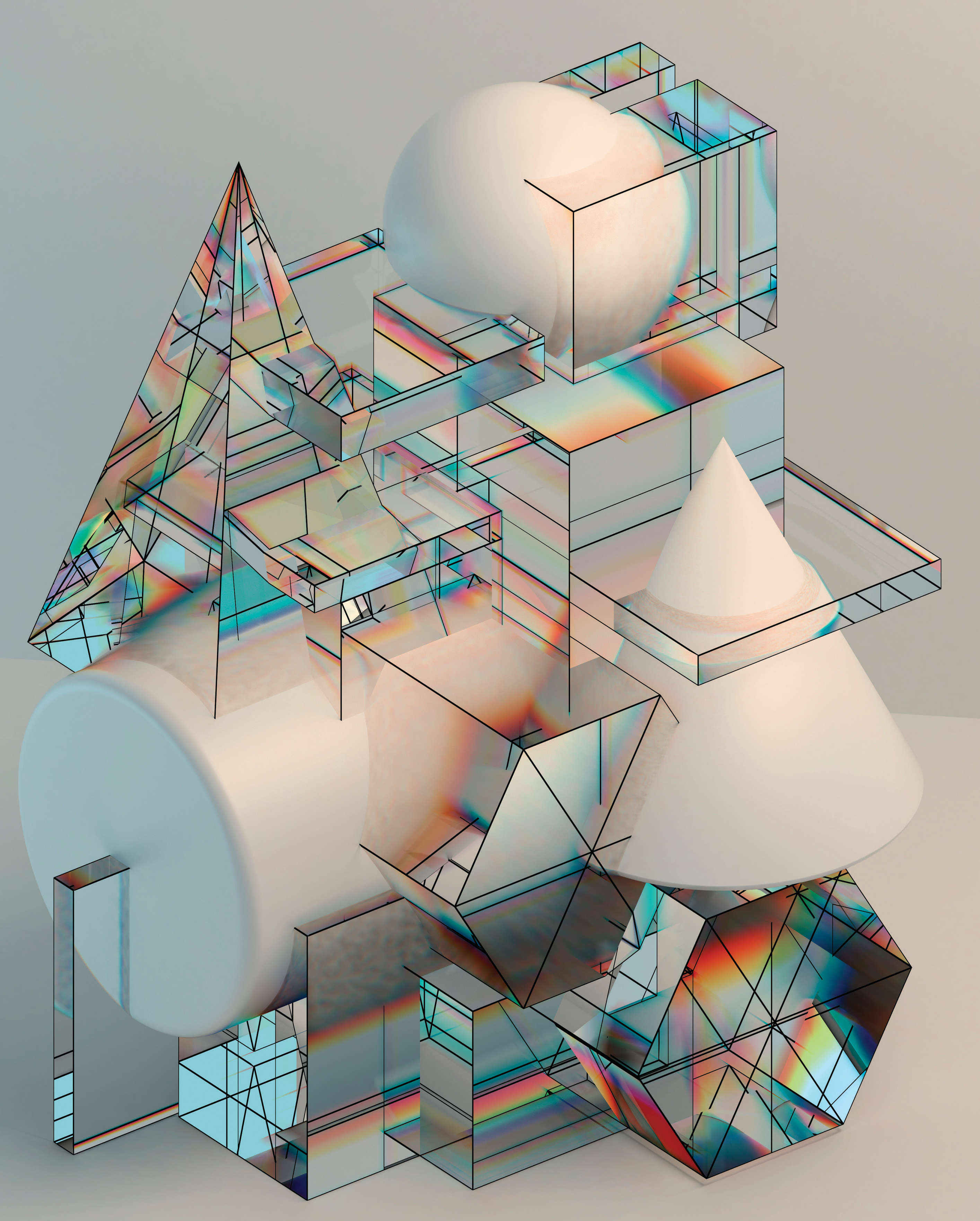
Design Goals

Like other scripting languages, Lua has dynamic types, dynamic data structures, garbage collection, and an `eval`-like functionality. Consider Lua's particular set of goals:

» key insights

- **What sets Lua apart from other scripting languages is its particular set of goals: simplicity, small size, portability, and embeddability.**
- **The entire implementation of Lua has 25,000 lines of C code; the binary for 64-bit Linux has 200k bytes.**
- **Since its inception, Lua was designed to interoperate with other languages.**

IMAGE BY BUG FISH



Simplicity. Lua aims to offer only a few powerful mechanisms that can address several different needs, instead of myriad specific language constructs, each tailored for a specific need. The Lua reference manual is small, with approximately 100 pages covering the language, its standard libraries, and the API with C;

Small size. The entire implementation of Lua consists of 25,000 lines of C code; the binary for 64-bit Linux has 200k bytes. Being small is important for both portability, as Lua must fit into a system before running there, and embedding, as it should not bloat the host application that embeds it;

Portability. Lua is implemented in ISO C and runs in virtually any system with as little as 300k bytes of memory. Lua runs in all mainstream systems and also on mainframes, inside OS kernels (such as the NetBSD kernel), and on "bare metal" (such as NodeMCU running on the ESP8266 microcontroller); and

Embeddability. Lua was designed since its inception to interoperate with other languages, both by extending—allowing Lua code to call functions written in a foreign language—and by embedding—allowing foreign code to call functions written in Lua.[8] Lua is thus implemented not as a standalone program but as a library with a C API. This library exports functions that create a new Lua state, `load` code into a state, call functions loaded into a state, access global variables in a state, and perform other basic tasks. The standalone Lua interpreter is a tiny application written on top of the library.

These goals have had a deep impact on our design of Lua. Portability restricts what the standard libraries can offer to what is available in ISO C, including date and time, file and string manipulation, and basic mathematical functions. Everything else must be provided by external libraries. Simplicity and small size restrict the language as a whole. These are the goals behind the economy of concepts for the language. Embeddability has a subtler influence. To improve embeddability, Lua favors mechanisms that can be represented naturally in the Lua-C API. For instance, Lua tries to avoid or reduce the use of special syntax for a new mechanism, as syntax is not accessible through an API. On the other hand, mechanisms exposed as functions are naturally mapped to the API.

Following the motto "mechanisms instead of policies" has a clear impact on simplicity and small size. It also affects embeddability by breaking complex concepts into simpler ones that are easier to represent in the API.

Lua supports eight data types: nil, boolean, number, string, userdata, table, function, and thread, which represents coroutines. The first five are no surprise. The last three give Lua its flavor and are the ones we discuss here. However, given the importance of embeddability in the design of Lua, we first briefly introduce the interface between Lua and its host language.

Figure 1. A C program using the Lua library.

```
#include <stdio.h>
#include "lauxlib.h"
#include "lualib.h"

int main (int argc, char **argv) {
  // create a new state
  lua_State *L = luaL_newstate();
  // load the standard libraries
  luaL_openlibs(L);
  // try to load the given file and then
  // call the resulting function
  if (luaL_loadfile(L, argv[1]) != LUA_OK ||
      lua_pcall(L, 0, 0, 0) != LUA_OK) {
    // some error occurred; print the error message
    fprintf(stderr, "lua: %s\n", lua_tostring(L, -1));
  }
  else {  // code ran successfully
    lua_getglobal(L, "pi");
    printf("pi: %f\n", lua_tonumber(L, -1));
  }
  lua_close(L);   // close the state
  return 0;
}
```

The Lua–C API

To illustrate the concept of embedding in Lua, consider a simple example of a C program using the Lua library. Take this tiny Lua script, stored in a file

```
pi = 4 * math.atan(1)
```

Figure 1 shows a C program that runs the script and prints the value of `pi`. The first task is to create a new state and populate it with the functions from the standard libraries (such as `math.atan`). The program then calls `luaL_loadfile` to `load` (precompile) the given source file into this state. In the absence of errors, this call produces a Lua function that is then executed by `lua_pcall`. If either `loadfile` or `pcall` raises an error, it produces an error message that is printed to the terminal. Otherwise, the program gets the value of the global variable `pi` and prints its value.

The data exchange among these API calls is done through an implicit stack in the Lua state. The call to `luaL_loadfile` pushes on the stack either a function or an error message. The call to `lua_pcall` pops the function from the stack and calls it. The call to `lua_getglobal` pushes the value of the global variable. The call to `lua_tonumber` projects the Lua value on top of the stack to a `double`. The stack ensures these values remain visible to Lua while being manipulated by the C code so they cannot be collected by Lua's garbage collector.

Besides the functions used in this simple example, the Lua–C API (or "C API" for short) offers functions for all kinds of manipulation of Lua values, including pushing C values (such as numbers and strings) onto the stack, calling functions defined by the script, and setting variables in the state.

Tables

"Table" is the Lua term for associative arrays, or "maps." A table is just a collection of entries, which are pairs ⟨key, value⟩.

Tables are the sole data-structuring mechanism in Lua. Nowadays, maps are available in most scripting

languages, as well as in several non-scripting ones, but in Lua maps are ubiquitous. Indeed, Lua programmers use tables not only for all kinds of data structures (such as records, arrays, lists, sets, and sparse matrices) but also for higher-level constructs (such as modules, objects, and environments).

Programmers implement records using tables whose indices are strings representing field names. Lua supports records with syntactic sugar, translating a field reference like `t.x` to a table-indexing operation `t["x"]`.

Lua offers constructors, expressions that create and initialize tables. The constructor `{}` creates an empty table. The constructor `{x=10,y=20}` creates a table with two entries, one mapping the string `"x"` to the integer 10, the other mapping `"y"` to 20. Programmers see this table as a record with fields `"x"` and `"y"`.

Programmers implement arrays with tables whose indices are positive integers. Constructors also support this usage. For example, the expression `{10,20,30}` creates a table with three entries, mapping 1 to 10, 2 to 20, and 3 to 30. Programmers see the table as an array with three elements.

Arrays have no special status in the semantics of Lua; they are just ordinary tables. However, arrays pervade programming. Therefore, implementation of tables in Lua gives special attention to their use as arrays. The internal representation of a table in Lua has two parts: an array and a hash.[7] If the array part has size N, all entries with integer keys between 1 and N are stored in the array part; all other entries are stored in the hash part. The keys in the array part are implicit and do not need to be stored. The size N of the array part is computed dynamically, every time the table has to rehash as the largest power of two such that at least half the elements in the array part will be filled. A generic access (such as `t[i]`) first checks whether `i` is an integer in the range $[1, N]$; this is the most common case and the one programmers expect to be fast. If so, the operation gets the value in the array; otherwise, it accesses the hash. When accessing record fields (such as `t.x`) the Lua core knows the key is a string and so skips the array test, going directly to the hash.

Lua offers exactly one general mechanism for each major aspect of programming: tables for data; functions for abstraction; and coroutines for control.

An interesting property of this implementation is that it gives sparse arrays for free. For instance, when a programmer creates a table with three entries at indices 5, 100, and 3421, Lua automatically stores them in the hash part, instead of creating a large array with thousands of empty slots.

Lua also uses tables to implement weak references. In languages with garbage collection, a weak reference is a reference to an object that does not prevent its collection as garbage.[10] In Lua, weak references are implemented in weak tables. A weak table is thus a table that does not prevent its contents from being collected. If a key or a value in an entry is collected, that entry is simply removed from the table; we discuss later how to signal that a table is weak. Weak tables in Lua also subsume ephemerons.[4]

Weak tables seem to contradict the motto "mechanisms instead of policies" because weak reference is a more basic concept than weak table. Weak tables would then be a policy, a particular way of using weak references. However, given the role of tables in Lua, it is natural to use them to support weak references without introducing yet another concept.

Functions

Lua supports first-class anonymous functions with lexical scoping, informally known as closures.[13] Several non-functional languages nowadays (such as Go, Swift, Python, and JavaScript) offer first-class functions. However, to our knowledge, none uses this mechanism as pervasively as Lua.

All functions in Lua are anonymous. This is not immediately clear in the standard syntax for defining a function

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

Nevertheless, this syntax is just syntactic sugar for an assignment of an anonymous function to a variable

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

Most dynamic languages offer some kind of `eval` function that evaluates a

piece of code produced at runtime. Instead of `eval`, Lua offers a `load` function that, given a piece of source code, returns a function equivalent to that code. We saw a variant of `load` in the C API in the form of `luaL_loadfile`. Consider the following piece of code

```
local id = 0
function genid ()
  id = id + 1
  return id
end
```

When one loads it, the function `load` returns an anonymous function equivalent to the following code

```
function ()
  local id = 0
  function genid ()
    id = id + 1
    return id
  end
end
```

So, if a programmer loads Lua code stored in a string and then calls the resulting function, the programmer gets the equivalent of `eval`.

We use the term "chunk" to denote a piece of code fed to `load` (such as a source file). Chunks are the compilation units of Lua. When a programmer uses Lua in interactive mode, the Read-Eval-Print Loop (REPL) handles each input line as a separate chunk.

The function `load` simplifies the semantics of Lua in two ways: First, unlike `eval`, `load` is pure and total; it has no side effects and it always returns a value, either a function or an error message; second, it eliminates the distinction between "global" code and "function" code, as in the previous chunk of code. The variable `id`, which in the original code appears outside any function, is seen by Lua as a local variable in the enclosing anonymous function representing the script. Through lexical scoping, `id` is visible to the function `genid` and preserves its value between successive calls to that function. Thus, `id` works like a static variable in C or a class variable in Java.

Exploring Tables and Functions

Despite their apparent simplicity—or because of it—tables and functions form a basis for several other mechanisms in Lua, including modules, object-oriented programming, and exception handling. We now discuss some of them, emphasizing how they contribute to Lua's design goals.

Modules. The construction of modules in Lua is a nice example of the use of first-class functions and tables as a basis for other mechanisms. At runtime, a module in Lua is a regular table populated with functions, as well as possibly other values (such as constants). Consider this Lua fragment

```
print(math.sin(math.pi/6))
   --> 0.5
```

Abstractly, programmers read this code as calling the `sin` function from the standard math module, using the constant `pi` from that same module. Concretely, the language sees `math` as a variable (created when Lua loaded its standard libraries) containing a reference to a table. That table has an entry with the key `"sin"` containing the sine function and an entry `"pi"` with the value of π.

Figure 2. A simple module in Lua.

```
local M = {}

function M.new (x, y)
  return {x = x, y = y}
end

function M.add (u, v)
  return M.new(u.x+v.x, u.y+v.y)
end

function M.norm (v)
  return math.sqrt(v.x^2 + v.y^2)
end

return M
```

Figure 3. A module in Lua using environments.

```
local sqrt = math.sqrt
local _ENV = {}

function new (x, y)
  return {x = x, y = y}
end

function add (u, v)
  return new(u.x+v.x, u.y+v.y)
end

function norm (v)
  return sqrt(v.x^2 + v.y^2)
end

return _ENV
```

Statically, a module is simply the chunk that creates its corresponding table. Figure 2 shows a standard idiom for defining a simple module in Lua. The code creates a table in the local variable `M`, populates the table with some functions, and returns that table. Recall that Lua loads any chunk as the body of an enclosing anonymous function; this is how one should read that code. The variable `M` is local to that enclosing function and the final statement returns from that function.

Once defined in a file `mymodule.lua`, a programmer can use that module with code like this[a]

```
local vec = require "mymodule"
print(vec.norm(vec.new(10, 10)))
   --> 14.142135623731
```

In it, `require` is a regular function from the standard library; when the single argument to a function is a literal string, the code can omit the parentheses in the call. If the module is not already loaded, `require` searches for an appropriate source for the given name (such as by looking for files in a list of paths), then loads and runs that code, and finally returns what the code returns. In this example, `require` returns the table `M` created by the chunk.

Lua leverages tables, first-class functions, and `load` to support modules. The only addition to the language is the function `require`. This economy is particularly relevant for an embedded language like Lua. Because `require` is a regular function, it cannot create local variables in the caller's scope. Thus, in the example using `"mymodule"`, the programmer had to define explicitly the local variable `vec`. Yet this limitation gives programmers the ability to give a local name to the module.

On the one hand, the construction of modules in Lua is not as elegant as a dedicated language mechanism could be, with explicit import and export lists and other refinements, as in the "import machinery" in Python.[12] On the other hand, this construction has a clear semantics that requires no

a To test these pieces of code interactively, remove the `local` from the variable initializations. In interactive mode, Lua loads each line as an independent chunk. A local variable is thus visible only in the line where it was defined.

further explanation. It also has an inexpensive implementation. Finally, and also quite important, it has an easy integration with the C API: One can easily create modules in C; create mixed modules with some functions defined in Lua and others in C; and for C code call functions inside modules. The API needs no additional mechanisms to do these tasks; all it needs is the existing Lua mechanisms to manipulate tables and functions.

Environments. Local variables in Lua follow a strict lexical scoping discipline. A local variable can be accessed only by code that is lexically written inside its scope. Lexical scoping implies that local variables are one of the few constructions that do not cross the C API, as C code cannot be lexically inside Lua code.

A program in Lua can be composed of multiple chunks (such as multiple modules) loaded independently. Lexical scoping implies that a module cannot create local variables for other chunks. Variables like `math` and `require`, created by the standard libraries, should thus be created as global variables. However, using global variables in a large program can easily lead to overly complex code, entangling apparently unrelated parts of a program. To circumvent this conflict, Lua does not have global variables built into the language. Instead, it offers a mechanism of environments that, by default, gives the equivalent of global variables. Nevertheless, as we show later in this article, environments allow other possibilities.

Recall that any chunk of code in Lua is compiled as if inside an anonymous function. Environments add two simple rules to this translation: First, the enclosing anonymous function is compiled as if in the scope of a local variable named `_ENV`; and second, any free variable `id` in the chunk is translated to `_ENV.id`. For example, Lua loads the chunk `print(v)` as if it was written like this

```
local _ENV = <<some given value>>
return function ()
    _ENV.print(_ENV.v)
end
```

By default, `load` initializes `_ENV` with a fixed table, called the global environment. All chunks thus share this same environment by default, giving the illusion of global variables; in the chunk just mentioned, both `v` and `print` refer to fields in that table and thus behave as global variables. However, both `load` and the code being loaded can modify `_ENV` to any other value. The `_ENV` mechanism allows different scripts to have different environments, functions to be called with different environments, and other variations.

The translation of free variables needs semantic information to determine whether a variable is free. Nevertheless, the translation itself is purely syntactical. In particular, `_ENV` is a regular variable, needing no special treatment by the compiler. The programmer can assign new values to `_ENV` or declare other variables with that name. As an example, consider this fragment

```
do
  local _ENV = {}
  ...
end
```

Inside the `do` block, all free variables refer to fields in the new table `_ENV`. Outside the block, all free variables refer to the default environment.

A more typical use of `_ENV` is for writing modules. Figure 3 shows how to rewrite the simple module of Figure 2 using environments. In the first line, where the code "imports" a function from the `math` module, the environment is still the default one. In the second line, the code sets the environment to a new table that will represent the module. The code then defines the module components directly as free variables; instead of `M.norm`, it uses only `norm`, which Lua translates to `_ENV.norm`. The code ends the module with `return _ENV`.

This method for writing modules has two benefits: First, all external functions and modules must be explicitly imported right at the start; and second, a module cannot pollute the global space by mistake.

Object-oriented programming. Support for object-oriented programming in Lua follows the pattern we have been seeing in this article: It tries to build upon tables and functions, adding only the minimum necessary to the language.

Lua uses a two-tier approach to object-oriented programming. The first is implemented by Lua and the second by programmers on top of the first one. The first tier is class-based. Both objects and classes are tables, and the relation "instance of" is dynamic. Userdata, which represents C values in Lua, can also play the role of objects. Classes are called metatables. In this first tier, a class can define only methods for the standard operators (such as addition, subtraction, and concatenation). These methods are called metamethods.

Figure 4 illustrates how a programmer would use this basic mechanism to perform arithmetic on 2D vectors. The code starts with a table `mt` that would be the metatable for the vectors. The code then defines a function `newVector` to create 2D vectors. Vectors are tables with two fields, `x` and `y`. The standard function `setmetatable` establishes the "instance of" relation

Figure 4. An example of metatables.

```
local mt = {}

function newVector (x, y)
  local p = {x = x, y = y}
  setmetatable(p, mt)
  return p
end

function mt.__add (p1, p2)
  return newVector(p1.x + p2.x, p1.y + p2.y)
end

-- example of use
A = newVector(10, 20)
B = newVector(20, -40)
C = A + B
print(C.x, C.y)      --> 30    -20
```

between a new vector and `mt`. Next, the code defines the metamethod `mt.__add` to implement the addition operator for vectors. The code then creates two vectors, `A` and `B`, and adds them to create a new vector `C`. When Lua tries to evaluate `A+B`, it does not know how to add tables and so checks for an `__add` entry in `A`'s metatable. Given that it finds that entry, Lua calls the function stored there—the metamethod—passing the original operands `A` and `B` as arguments.

The metamethod for the indexing operator `[]` offers a form of delegation in Lua. Lua calls this metamethod, named `__index`, whenever it tries to retrieve the value of an absent key from a table. (For userdata, Lua calls that metamethod for all keys.) For the indexing operation, Lua allows the metamethod to be a function or a table. When `__index` is a table, Lua delegates to that table all access for an index that is absent in the original table, as illustrated by this code fragment

```
Proto = {x = 0, y = 0}
obj = {x = 10}
mt = {__index = Proto}
setmetatable(obj, mt)
print(obj.x) --> 10
print(obj.y) --> 0
```

In the second call to `print`, Lua cannot find the key `"y"` in `obj` and so delegates the access to `Proto`. In the first `print`, as `obj` has a field `"x"`, the access is not delegated.

With tables, functions, and delegation, we have almost all we need for the second tier, which is based on prototypes. In it, programmers represent objects also by tables or userdata. Each object can have a prototype, from which it inherits methods and fields. The prototype of an object `obj` is the object stored in the `__index` field of the metatable of `obj`. One can then write `obj.foo(x)`, and Lua will retrieve the method `foo` from the object's prototype, through delegation.

However, if we stopped here, there would be a flaw in the support for object-oriented programming in Lua. After finding and calling the method in the object's prototype, there would be no way for the method to access the original object, which is the intended receiver. Lua solves this problem through syntactic sugar. Lua translates a "method" definition like

```
function Proto:foo (x)
  ...
end
```

to a function definition:

```
function Proto.foo (self, x)
  ...
end
```

Likewise, Lua translates a "method" call `obj:foo(x)` to `obj.foo(obj,x)`. When the programmer defines a "method"—a function using the colon syntax—Lua adds a hidden parameter `self`. When the programmer calls a "method" using the colon syntax, Lua provides the receiver as the argument to the `self` parameter. There is no need to add classes, objects, or methods to the language, merely syntactic sugar.

Figure 5. A simple prototype-based design in Lua.

```
local Account = {bal = 0}
local mt = {__index = Account}

function Account:new ()
  local obj = {}
  setmetatable(obj, mt)
  return obj
end

function Account:deposit (amount)
  self.bal = self.bal + amount
end
function Account:balance ()
  return self.bal
end

return Account
```

Figure 5 illustrates these concepts. First the code creates a prototype, the table `Account`. The code then creates a table `mt` to be used as the metatable for instances of `Account`. It then adds three methods to the prototype: one for creating instances, one for making deposits, and one for retrieving the account's balance. Finally, it returns the prototype as the result of this module.

Assuming the module is in the file `Account.lua`, the following lines exercise the code

```
Account = require "Account"
acc = Account:new()
acc:deposit(1000)
print(acc:balance()) -->
1000
```

First, the code requires the module, then it creates an account; `acc` will be an empty table with `mt` as its metatable. De-sugared, the next line reads as `acc.deposit(acc,1000)`. The table `acc` does not have a `deposit` field, so Lua delegates that access to the table in the metatable's `__index` field. The result of the access is the function `Account.deposit`. Lua then calls that function, passing `acc` as the first argument (`self`) and 1000 as the second argument (`amount`). Inside the function, Lua will again delegate the access `self.bal` to the prototype because `acc` does not yet have a field `bal`. In subsequent calls to balance, Lua will find a field `bal` in the table `acc` and use that value. Distinct accounts thus have separate balances but share all methods.

The access to a prototype in the metatable's `__index` is a regular access, meaning prototypes can be chained. As an example, suppose the programmer adds the following lines to the previous example

```
Object = {name = "no name"}
setmetatable(Account,
    {__index = Object})
```

When Lua evaluates `acc.name`, the table `acc` does not have a `name` key, so Lua tries the access in its prototype, `Account`. That table also does not have that key, so Lua goes to `Account`'s prototype, the table `Object`, where it finally finds a `name` field.

Figure 6. Accounts with private fields.

```
local bal = {}
setmetatable(bal, {__mode = "k"})

local Account = {}
local mt = {__index = Account}

function Account:new ()
  local obj = {}
  setmetatable(obj, mt)
  bal[obj] = 0
  return obj
end

function Account:deposit (amount)
  bal[self] = bal[self] + amount
end

function Account:balance ()
  return bal[self]
end

return Account
```

The programmer can keep the balances private by storing them outside the object table, as shown in Figure 6. The key difference between this version and the one in Figure 5 is the use of `bal[self]` instead of `self.bal` to denote the balance of an account. The table `bal` is what we call a dual table. The call to `setmetatable` in the second line causes this table to have weak keys, thus allowing an account to be collected when there are no other references to it in the program. The fact that `bal` is local to the module ensures no code outside that module can see or tamper with an account's balance, a technique that is handy whenever one needs a private field in a structure.

An evaluation of Lua's support for object-oriented programming is not very different from the evaluation of the other mechanisms we have discussed so far. On the one hand, object-oriented features in Lua are not as easy to use as in other languages that offer specific constructs for the task. In particular, the colon syntax can be somewhat confusing, mainly for programmers who are new to Lua but have some experience with another object-oriented language. Lua needs that syntax because of its economy of concepts that avoids introducing the concept of method when the existing concept of function will suffice.

On the other hand, the semantics of objects in Lua is simple and clear. Also, the implementation of objects in Lua is flexible. Because method selection and the variable `self` are independent, Lua does not need additional mechanisms to call methods from other classes (such as "super"). Finally, this design is friendly to the C API. All it needs is basic manipulation of tables and functions, plus the standard function `setmetatable`. Lua programmers can implement prototypes in Lua and create userdata instances in C, create prototypes in C and instances in Lua, and define prototypes with some methods implemented in Lua and others in C. All these pieces work together seamlessly.

Exception handling. Exception handling in Lua is another mechanism that relies on the flexibility of functions. Several languages offer a `try-catch` construction for exception handling; any exception in the code inside a `try` clause jumps to a corresponding `catch` clause. Lua does not offer such a construction, mainly because of the C API.

More often than not, exceptions in a script are handled by the host application. A syntactic construction like `try-catch` is not easily mapped into an API with a foreign language. Instead, the C API packs exception-handling functionality into the higher-order function `lua_pcall` ("protected call") we discussed when we visited the C API earlier in this article. The function `pcall` receives a function as an argument and calls that function. If the provided function terminates without errors, `pcall` returns true; otherwise, `pcall` catches the error and returns false plus an error object, which is any value given when the error was raised. Regardless of how `pcall` is implemented, it is exposed in the C API as a conventional function. The C API also offers a function to raise errors, called `lua_error`, whose only argument is the error object. The function error also appears in the C API as a regular function despite the fact that it never returns.

Both `lua_pcall` and `lua_error` are reflected into Lua via the standard library. In languages that support `try-catch`, typical exception-handling code looks like this

```
try {
  <<protected code>>
}
catch (errobj) {
  <<exception handling>>
}
```

The equivalent code in Lua is like this

```
local ok, errobj =
pcall(function ()
  <<protected code>>
end)

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

In this translation, anonymous functions with proper lexical scoping play a central role. Except for statements that invoke escape continuations (such as `break` and `return`), everything else can be written inside the protected code as if written in the regular code.

The use of `pcall` for exception handling has pros and cons similar to those for modules. On the one hand, the code may not look as elegant as in other languages that support the traditional `try`. On the other hand, it has a clear semantics. In particular, questions like "What happens with exceptions inside the catch clause?" have an obvious answer. Moreover, it has a clear and easy integration with the C API; it is exposed through conventional

Figure 7. A simple example of a coroutine in Lua.

```
co = coroutine.create(function (x)
    print(x)   --> 10
    x = coroutine.yield(20)
    print(x)   --> 30
    return 40
end)

print(coroutine.resume(co, 10))  --> 20
print(coroutine.resume(co, 30))  --> 40
```

functions; and Lua programs can raise errors in Lua and catch them in C and raise errors in C and catch them in Lua.

Coroutines

Like associative arrays and first-class functions, coroutines are a well-established concept in programming. However, unlike tables and first-class functions, there are significant variations in how different communities implement coroutines.[2] Several of these variations are not equivalent, in the sense that a programmer cannot implement one on top of the other.

Coroutines in Lua are like cooperative multithreading and have the following distinguishing properties:

First-class values. Lua programmers can create coroutines anywhere, store them in variables, pass them as parameters, and return them as results. More important, they can resume coroutines anywhere;

Suspend execution. They can suspend their execution from within nested functions. Each coroutine has its own call stack, with a semantics similar to collaborative multithreading. The entire stack is preserved when the coroutine yields;

Asymmetric. Symmetric coroutines offer a single control-transfer operation that transfers control from the running coroutine to another given coroutine. Asymmetric coroutines, on the other hand, offer two control-transfer operations, `resume` and `yield`, that work like a call–return pair; and

Equivalent to one-shot continuations.[2] Despite this equivalence, coroutines offer one-shot continuations in a format that is more natural for a procedural language due to its similarity to multithreading.

Figure 7 illustrates the life cycle of a coroutine in Lua. The program prints 10, 20, 30, and 40, in that order. It starts by creating a coroutine `co`, giving an anonymous function as its body. That operation returns only a handle to the new coroutine, without running it. The program then resumes the coroutine for the first time, starting the execution of its body. The parameter `x` receives the argument given to `resume`, and the program prints 10. The coroutine then yields, causing the call to `resume` to return the value 20, the argument given to `yield`. The program then resumes the coroutine again, making `yield` return 30, the value given to `resume`. The coroutine then prints 30 and finishes, causing the corresponding call to `resume` to return 40, the value returned by the coroutine.

Coroutines are not as widely used in Lua as tables and functions. Nevertheless, when required, coroutines play a pivotal role, due to their capacity for turning the control flow of a program inside out.

An important use of coroutines in Lua is for implementing cooperative multithreading. Games typically exploit this feature, because they need to be in control to remain responsive at interactive rates. Each character or object in a game has its own script running in a separate coroutine. Each script is typically a loop that, at each iteration, updates the character's state and then yields. A simple scheduler resumes all live coroutines at each game update.

Another use of coroutines is in tackling the "who-is-the-boss" problem. A typical issue with scripting languages is the decision whether to embed or to extend. When programmers embed a scripting language, the host is the boss, that is, the host program, written in the foreign language, has the main loop of the program and calls functions written in the scripting language for particular tasks. When programmers extend a scripting language, the script is the boss; programmers then write libraries for it in the foreign language, and the main loop of the program is in the script.

Embedding and extending both have advantages and disadvantages, and the Lua–C API supports them equally. However, external code can be less forgiving. Suppose a large, monolithic application contains some useful functionality for a particular script. The programmer wants to write the script as the boss, calling functions from that external application. However, the application itself assumes *it* is the boss. Moreover, it may be difficult to break the application into individual functions and offer them as a coherent library to the script.

Coroutines offer a simpler design. The programmer modifies the application to create a coroutine with the script when it starts; every time

the application needs an input, it resumes that coroutine. That is the only change the programmer needs to make in the application. The script, for its part, also looks like a regular program, except it yields when it needs to send a command to the application. The control flow of the resulting program progresses as follows: The application starts, creates the coroutine, does its own initialization, and then waits for input by resuming the coroutine. The coroutine then starts running, does its own initialization, and performs its duties until it needs some service from the application. At this point, the script yields with a request, the call to `resume` made by the application returns, and the application services the given request. The application then waits for the next request by resuming the script again.

Presentation of coroutines in the C API is clearly more challenging than presentation of functions and tables. C code can create and resume coroutines without restrictions. In particular, resuming works like a regular function call: It (re) activates the given coroutine when called and returns when the coroutine yields or ends. However, yielding also poses a problem. Once a C function yields, there is no way to later return the control to that point in the function. The API offers two ways to circumvent this restriction: The first is to yield in a tail position: When the coroutine resumes, it goes straight to the calling Lua function. The second is to provide a continuation function when yielding. In this way, when the coroutine resumes, the control goes to the continuation function, which can finish the task of the original function.

We can see again in the API the advantages of asymmetric coroutines for a language like Lua. With symmetric coroutines, all transfers would have the problems that asymmetric coroutines have only when yielding. In our experience, resumes from C are much more common than yields.

Conclusion

Every design involves balancing conflicting goals. To address the conflicts, designers need to prioritize their goals. This is clearly true of the design of any programming language.

Lua has a unique set of design goals that prioritize simplicity, portability, and embedding. The Lua core is based on three well-known, proven concepts—associative arrays, first-class functions, and coroutines—all implemented with no artificial restrictions. On top of these components, Lua follows the motto "mechanisms instead of policies," meaning Lua's design aims to offer basic mechanisms to allow programmers to implement more complex features. For instance, in the case of modules, tables provide name spaces, lexical scoping provides encapsulation, and first-class functions allow exportation of functions. On top of that, Lua adds only the function `require` to search for and `load` modules.

Modularity in language design is nothing new.[11] For instance, it can be used to clarify the construction of a large application.[1] However, Lua uses modularity to keep its size small, breaking down complex constructions into existing mechanisms.

The motto "mechanisms instead of policies" also makes for a flexible language, sometimes too flexible. For instance, the do-it-yourself approach to classes and objects leads to proliferation of different, often incompatible, systems, but is handy when a programmer needs to adapt Lua to the class model of the host program.

Tables, functions, and coroutines as used in Lua have shown great flexibility over the years. Despite the language's continuing evolution, there has been little demand from programmers to change the basic mechanisms.

The lack of built-in complex constructions and minimalist standard libraries (for portability and small size) make Lua a language that is not as good as other scripting languages for writing "quick-and-dirty" programs. Many programs in Lua need an initial phase for programmers to set up the language, as a minimal infrastructure for object-oriented programming. More often than not, Lua is embedded in a host application. Embedding demands planning and the set-up of the language is typically integrated with its embedding. Lua's economy of concepts demands from programmers a deeper understanding of what they are doing, as most constructions are explicit in the code. This explicitness also allows such deeper understanding. We trust this is a blessing, not a curse. **C**

References

1. Cazzola, W. and Olivares, D.M. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing 4*, 3 (July 2016), 404–415.
2. de Moura, A.L and Ierusalimschy, R. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems 31*, 2 (Feb. 2009), 6.1–6.31.
3. Gamasutra. *Game Developer* magazine's 2011 Front Line Award, Jan. 13, 2012; https://www.gamasutra.com/view/news/129084/
4. Hayes, B. Ephemerons: A new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Atlanta, GA, Oct. 5–9). ACM, New York, 1997, 176–183.
5. Ierusalimschy, R. Programming with multiple paradigms in Lua. In *Proceedings of the 18th International Workshop on Functional and (Constraint) Logic Programming, LNCS, Volume 5979*. S. Escobar, Ed. (Brasilia, Brazil, June 28). Springer, Heidelberg, Germany, 2009, 5–13.
6. Ierusalimschy, R., de Figueiredo, L.H., and Celes, W. Lua—An extensible extension language. *Software: Practice and Experience 26*, 6 (June 1996), 635–652.
7. Ierusalimschy, R., de Figueiredo, L.H., and Celes, W. The evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, CA, June 9–10). ACM Press, New York, 2007, 2.1–2.26.
8. Ierusalimschy, R., de Figueiredo, L.H., and Celes, W. Passing a language through the eye of a needle. *Commun. ACM 54*, 7 (July 2011), 38–43.
9. International Organization for Standardization. ISO 2000. *International Standard: Programming Languages, C*. ISO/IEC9899: 1999(E).
10. Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook*. CRC Press, Boca Raton, FL, 2011.
11. Kats, L. and Visser, E. The Spoofax Language Workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, NV, Oct. 17–21). ACM Press, New York, 2010, 444–463.
12. The Python Software Foundation. *The Python Language Reference, 3.5 Edition*. The Python Software Foundation, 2015.
13. Sestoft, P. *Programming Language Concepts, Second Edition*. Springer, Cham, Switzerland, 2017.
14. Wikipedia. List of applications using Lua; https://en.wikipedia.org/w/index.php?title=List_of_applications_using_Lua&oldid=795421653

Roberto Ierusalimschy (roberto@inf.puc-rio.br) is an associate professor of computer science at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro, Brazil.

Luiz Henrique de Figueiredo (lhf@impa.br) is a researcher at IMPA, the Institute for Pure and Applied Mathematics in Rio de Janeiro, Brazil.

Waldemar Celes (celes@inf.puc-rio.br) is an associate professor of computer science at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro, Brazil.

Copyright held by the authors.
Publication rights licensed to ACM. $15.00

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/a-look-at-the-design-of-lua