DOI:10.1145/1965724.1965739

Article development led by acmqueue
queue.acm.org

**How the embeddability of Lua impacted its design.**

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

# Passing a Language Through the Eye of a Needle

SCRIPTING LANGUAGES ARE an important element in the current landscape of programming languages. A key feature of a scripting language is its ability to integrate with a system language.[7] This integration takes two main forms: *extending* and *embedding*. In the first form, you extend the scripting language with libraries and functions written in the system language and write your main program in the scripting language. In the second form, you embed the scripting language in a host program (written in the system language) so that the host can run scripts and call functions defined in the scripts; the main program is the host program. In this setting, the system language is usually called the host language.

Many languages (not necessarily scripting languages) support extending through a *foreign function interface* (FFI). An FFI is not enough to allow a function in the system language to do all that a function in the script can do. Nevertheless, in practice FFI covers most common needs for extending, such as access to external libraries and system calls. Embedding, on the other hand, is more difficult to support, because it usually demands closer integration between the host program and the script, and FFI alone does not suffice.

In this article we discuss how embeddability can impact the design of a language, and in particular how it impacted the design of Lua from day one. Lua[3,4] is a scripting language with a particularly strong emphasis on embeddability. It has been embedded in a wide range of applications and is a leading language for scripting games.[2]

## The Eye of a Needle

At first sight, the embeddability of a scripting language seems to be a feature of the implementation of its interpreter. Given any interpreter, we can attach an API to it to allow the host program and the script to interact. The design of the language itself, however, has a great influence on the way it can be embedded. Conversely, if you design a language with embeddability in mind, this mind-set will have a great influence on the final language.

The typical host language for most scripting languages is C, and APIs for these languages are therefore mostly composed of functions plus some types and constants. This imposes a natural but narrow restriction on the design of an API for a scripting language: it must offer access to language features through this eye of a needle. Syntactical constructs are particularly difficult to get through. For example, in a scripting language where methods must be written lexically inside their classes, the host language cannot add methods to a class unless the API offers suitable mechanisms.

Similarly, it is difficult to pass lexical scoping through an API, because host functions cannot be lexically inside scripting functions.

A key ingredient in the API for an embeddable language is an `eval` function, which executes a piece of code. In particular, when a scripting language is embedded, all scripts are run by the host calling `eval`. An `eval` function also allows a minimalist approach for designing an API. With an adequate `eval` function, a host can do practically anything in the script environment: it can assign to variables (`eval"a = 20"`), query variables (`eval"return a"`), call functions (`eval"foo(32,'stat')"`), and so on. Data structures such as arrays can be constructed and decomposed by evaluating proper code. For example, again assuming a hypothetical `eval` function, the C code shown in Figure 1 would copy a C array of integers into the script.

Despite its satisfying simplicity and completeness, an API composed of a single `eval` function has two drawbacks: it is too inefficient to be used intensively, because of the cost of parsing and interpreting a chunk at each interaction; and it is too cumbersome to use, because of the string manipulation needed to create commands in C and the need to serialize all data that goes through the API. Nevertheless, this approach is often used in real applications. Python calls it "Very High-Level Embedding."[8]

For a more efficient and easier-to-use API, we need more complexity. Besides an `eval` function for executing scripts, we need direct ways to call functions defined by scripts, to handle errors in scripts, to transfer data between the host program and the scripting environment, and so on. We will discuss these various aspects of an API for an embeddable language and how they have affected and been affected by the design of Lua, but first we discuss how the simple existence of such an API can affect a language.

Given an embeddable language with its API, it is not difficult to write a library in the host language that exports the API back into the scripting language. So, we have an interesting form of reflection, with the host language acting as a mirror. Several mechanisms in Lua use this technique. For example, Lua offers a function called `type` to query the type of a given value. This function is implemented in C outside the interpreter, through an external library. The library simply exports to Lua a C function (called `luaB _ type`) that calls the Lua API to get the type of its argument.

On the one hand, this technique simplifies the implementation of the interpreter; once a mechanism is available to the API, it can easily be made available to the language. On the other hand, it forces language features to pass through the eye of the needle, too. We will see a concrete example of this trade-off when we discuss exception handling.

## Control

The first problem related to control that every scripting language must solve is the "who-has-the-main-function" problem. When we use the scripting language embedded in a host, we want the language to be a library, with the main function in the host. For many applications, however, we want the language as a standalone program with its own internal main function.

Lua solves this problem with the use of a separate standalone program. Lua itself is entirely implemented as a library, with the goal of being embedded in other applications. The `lua` command-line program is just a small application that uses the Lua library as any other host to run pieces of Lua code. The code in Figure 2 is a barebones version of this application. The real application, of course, is longer than that, as it has to handle options, errors, signals, and other real-life details, but it still has fewer than 500 lines of C code.

Although function calls form the

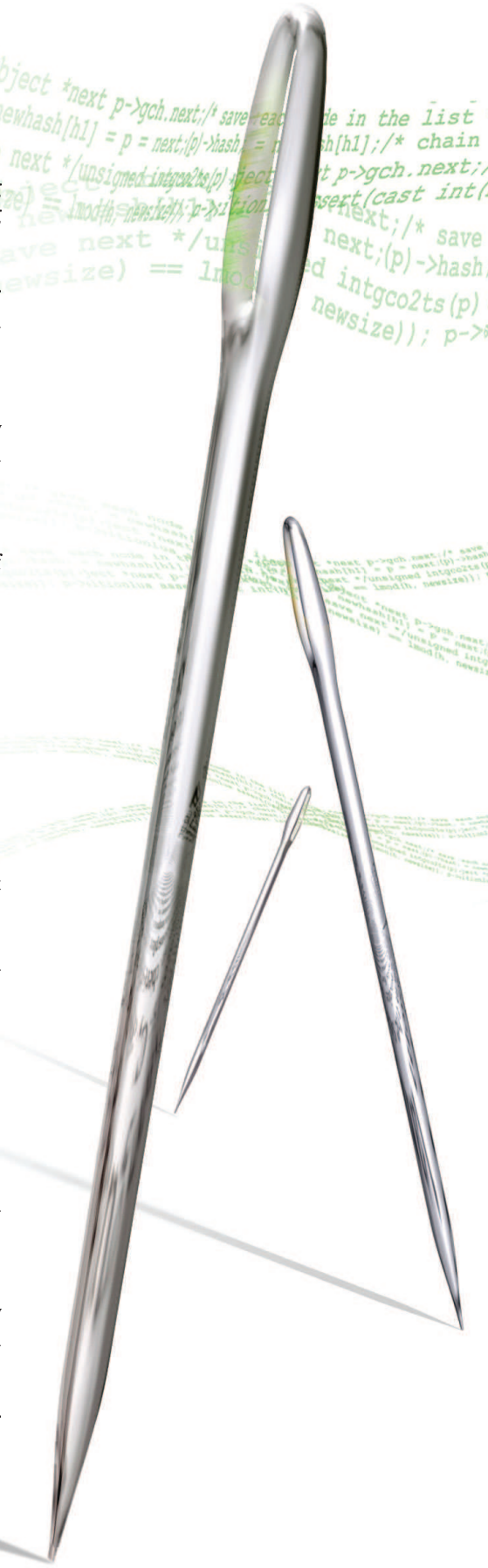

ILLUSTRATION BY J.F. PODEVIN

bulk of control communication between Lua and C, there are other forms of control exposed through the API: iterators, error handling, and coroutines. Iterators in Lua allow constructions such as the following one, which iterates over all lines of a file:

```
for line in io.lines(file) do
   print(line)
end
```

Although iterators present a new syntax, they are built on top of first-class functions. In our example, the call `io.lines(file)` returns an *iteration function*, which returns a new line from the file each time it is called. So, the API does not need anything special to handle iterators. It is easy both for Lua code to use iterators written in C (as is the case of `io.lines`) and for C code to iterate using an iterator written in Lua. For this case there is no syntactic support; the C code must do explicitly all that the `for` construct does implicitly in Lua.

Error handling is another area where Lua has suffered a strong influence from the API. All error handling in Lua is based on the `longjump` mechanism of C. It is an example of a feature exported from the API to the language.

The API supports two mechanisms for calling a Lua function: u*nprotected* and *protected*. An unprotected call does not handle errors: any error during the call `long jumps` through this code to land in a protected call farther down the call stack. A protected call sets a recovery point using `setjmp`, so that any error during the call is captured; the call always returns with a proper error code. Such protected calls are very important in an embedded scenario where a host program cannot afford to abort because of occasional errors in a script. The bare-bones application just presented uses `lua_pcall` (protected call) to call each compiled line in protected mode.

The standard Lua library simply exports the protected-call API function to Lua under the name of `pcall`. With `pcall`, the equivalent of a *try-catch* in Lua looks like this:

```
local ok, errorobject = pcall(function()
  --here goes the protected code
  ...
end)
if not ok then
  --here goes the error handling code
  --(errorobject has more information about
  the error)
  ...
end
```

This is certainly more cumbersome than a try-catch primitive mechanism built into the language, but it has a perfect fit with the C API and a very light implementation.

The design of coroutines in Lua is another area where the API had a great impact. Coroutines come in two flavors: *symmetric* and *asymmetric*.[1] Symmetric coroutines offer a single control-transfer primitive, typically called *transfer*, that acts like a *goto*: it can transfer control from any coroutine to any other. Asymmetric coroutines offer two control-transfer primitives, typically called *resume* and *yield*, that act like a pair *call–return*: a `resume` can transfer control to any other coroutine; a `yield` stops the current coroutine and goes back to the one that resumed the one yielding.

It is easy to think of a coroutine as a call stack (a *continuation*) that encodes which computations a program must do to finish that coroutine. The *transfer* primitive of symmetric coroutines corresponds to replacing the entire call stack of the running coroutine by the call stack of the transfer target. On the other hand, the *resume* primitive adds the target stack on top of the current one.

A symmetric coroutine is simpler than an asymmetric one but poses a big problem for an embeddable language such as Lua. Any active C function in a script must have a corresponding activation register in the C stack. At any point during the execution of a script, the call stack may have a mix of C functions and Lua functions. (In particular, the bottom of the call stack always has a C function, which is the host program that initiated the script.) A program cannot remove these C entries from the call stack, however, because C does not offer any mechanism for manipulating its call stack. Therefore, the program cannot make any transfer.

**Figure 1. Passing an array through an API with *eval*.**

```
void copy (int ar[], int n) {
      int i;
      eval("ar = {}"); /* create an empty array */
      for (i =0; i <n; i++){
        char buff[100];
        sprintf(buff, "ar[%d] = %d", i + 1, ar[i]);
        eval(buff); /* assign i-th element */
      }
}
```

**Figure 2. The bare-bones Lua application.**

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

int main (void) {
  char line[256];
  lua_State *L = luaL_newstate(); /* create a new state */
  luaL_openlibs(L);          /* open the standard libraries */

  /* reads lines and executes them */
  while (fgets(line, sizeof(line), stdin) != NULL) {
    luaL_loadstring(L, line); /* compile line to a function */
    lua_pcall(L, 0, 0, 0);             /* call the function */
  }

  lua_close(L);
  return 0;
}
```

Asymmetric coroutines do not have this problem, because the *resume* primitive does not affect the current stack. There is still a restriction that a program cannot yield *across* a C call—that is, there cannot be a C function in the stack between the `resume` and the *yield*. This restriction is a small price to pay for allowing portable coroutines in Lua.

## Data

One of the main problems with the minimalist *eval* approach for an API is the need to serialize all data either as a string or a code segment that rebuilds the data. A practical API should therefore offer other more efficient mechanisms to transfer data between the host program and the scripting environment.

When the host calls a script, data flows from the host program to the scripting environment as arguments, and it flows in the opposite direction as results. When the script calls a host function, we have the reverse. In both cases, data must be able to flow in both directions. Most issues related to data transfer are therefore relevant both for embedding and extending.

To discuss how the Lua–C API handles this flow of data, let's start with an example of how to extend Lua. Figure 3 shows shows the implementation of function `io.getenv`, which accesses environment variables of the host program.

For a script to be able to call this function, we must *register* it into the script environment. We will see how to do this in a moment; for now, let us assume that it has been registered as a global variable `getenv`, which can be used like this:

```
print(getenv("PATH"))
```

The first thing to note in this code is the prototype of `os _ getenv`. The only parameter of that function is a Lua state. The interpreter passes the actual arguments to the function (in this example, the name of the environment variable) through a data structure inside this state. This data structure is a stack of Lua values; given its importance, we refer to it as *the stack*.

When the Lua script calls `getenv`, the Lua interpreter calls `os _ getenv` with the stack containing only the arguments given to `getenv`, with the first argument at position 1 in the stack. The first thing `os _ getenv` does is to call `luaL _ checkstring`, which checks whether the Lua value at position 1 is really a string and returns a pointer to the corresponding C string. (If the value is not a string, `luaL _ checkstring` signals an error using a `longjump`, so that it does not return to `os _ getenv`.)

Next, the function calls `getenv` from the C library, which does the real work. Then it calls `lua _ pushstring`, which converts the C string value into a Lua string and pushes that string onto the stack. Finally, `os _ getenv` returns 1. This return tells the Lua interpreter how many values on the top of the stack should be considered the function results. (Functions in Lua may return multiple results.)

Now let's return to the problem of how to register `os _ getenv` as `getenv` in the scripting environment. One simple way is by changing our previous example of the basic standalone Lua program as follows:

```
lua _ State *L = luaL _ newstate();
/* creates a new state */
luaL _ openlibs(L);
/* opens the standard libraries */

+ lua _ pushcfunction(L, os _ getenv);
+ lua _ setglobal(L, "getenv");
```

The first added line is all the magic we need to extend Lua with host functions. Function `lua _ pushcfunction` receives a pointer to a C function and pushes on the stack a (Lua) function that, when called, calls its corresponding C function. Because functions in Lua are first-class values, the API does not need extra facilities to register global functions, local functions, methods, and so forth. The API needs only the single injection function `lua _ pushcfunction`. Once created as a Lua function, this new value can be manipulated just as any other Lua value. The second added line in the new code calls `lua _ setglobal` to set the value on the top of the stack (the new function) as the value of the global variable `getenv`.

Besides being first-class values, functions in Lua are always anonymous. A declaration such as

```
function inc (x) return x + 1 end
```

is syntactic sugar for an assignment:

```
inc = function (x) return x + 1 end
```

The API code we used to register function `getenv` does exactly the same thing as a declaration in Lua: it creates an anonymous function and assigns it to a global variable.

In the same vein, the API does not need different facilities to call different kinds of Lua functions, such as global functions, local functions, and methods. To call any function, the host first uses the regular data-manipulation facilities of the API to push the function onto the stack, and then pushes the arguments. Once the function (as a first-class value) and the arguments are in the stack, the host can call it with a single API primitive, regardless of where the function came from.

One of the most distinguishing features of Lua is its pervasive use of tables. A table is essentially an associative array. Tables are the only data-structure mechanisms in Lua, so they play a much larger role than in other languages with similar constructions. Lua uses tables not only for all its data structures (records and arrays among others), but also for other language mechanisms, such as modules, objects, and environments.

The example in Figure 4 illustrates the manipulation of tables through the

**Figure 3. A simple C function.**

```
static int os_getenv (lua_State *L) {
  const char *varname = luaL_checkstring(L, 1);
  const char *value = getenv(varname);
  lua_pushstring(L, value);
  return 1;
}
```

API. Function os_environ creates and returns a table with all environment variables available to a process. The function assumes access to the environ array, which is predefined in POSIX systems; each entry in this array is a string of the form NAME=VALUE, describing an environment variable.

The first step of os_environ is to create a new table on the top of the stack by calling lua_newtable. Then the function traverses the array environ to build a table in Lua reflecting the contents of that array. For each entry in environ, the function pushes the variable name on the stack, pushes the variable value, and then calls lua_settable to store the pair in the new table. (Unlike lua_pushstring, which assumes a zero-terminated string, lua_pushlstring receives an explicit length.)

Function lua_settable assumes that the key and the value for the new entry are on the top of the stack; the argument -3 in the call tells where the table is in the stack. (Negative numbers index from the top, so -3 means three slots from the top.)

Function lua_settable pops both the key and the value, but leaves the table where it was in the stack. Therefore, after each iteration, the table is back on the top. The final return1 tells Lua that this table is the only result of os_environ.

A key property of the Lua API is that it offers no way for C code to refer directly to Lua objects; any value to be manipulated by C code must be on the stack. In our last example, function os_environ creates a Lua table, fills it with some entries, and returns it to the interpreter. All the time, the table remains on the stack.

We can contrast this approach with using some kind of C type to refer to values of the language. For example, Python has the type PyObject; JNI (Java Native Interface) has jobject. Earlier versions of Lua also offered something similar: a lua_Object type. After some time, however, we decided to change the API.[6]

The main problem of a lua_Object type is the interaction with the garbage collector. In Python, the programmer is responsible for calling macros such as Py_INCREF and DECREF to increment and decrement the reference count of objects being manipulated by the API. This explicit counting is both complex and error prone. In JNI (and in earlier versions of Lua), a reference to an object is valid until the function where it was created returns. This approach is simpler and safer than a manual counting of references, but the programmer loses control of the lifetime of objects. Any object created in a function can be released only when the function returns. In contrast, the stack allows the programmer to control the lifetime of any object in a safe way. While an object is in the stack, it cannot be collected; once out of the stack, it cannot be manipulated. Moreover, the stack offers a natural way to pass parameters and results.

The pervasive use of tables in Lua has a clear impact on the C API. Anything in Lua represented as a table can be manipulated with exactly the same operations. As an example, modules in Lua are implemented as tables. A Lua module is nothing more than a table containing the module functions and occasional data. (Remember, functions are first-class values in Lua.) When you write something like math.sin(x), you think of it as calling the sin function from the math module, but you are actually calling the contents of field "sin" in the table stored in the global variable math. Therefore, it is very easy for the host to create modules, to add functions to existing modules, to "import" modules written in Lua, and the like.

Objects in Lua follow a similar pattern. Lua uses a prototype-based style for object-oriented programming, where objects are represented by tables. Methods are implemented as functions stored in prototypes. Similarly to modules, it is very easy for the host to create objects, to call methods, and so on. In class-based systems, instances of a class and its subclasses must share some structure. Prototype-based systems do not have this requirement, so host objects can inherit behavior from scripting objects and vice versa.

**Figure 4. A C function that returns a table.**

```
extern char **environ;
static int os_environ (lua_State *L) {
  int i;

  /* push a new table onto the stack */
  lua_newtable(L);

  /* repeat for each environment variable */
  for (i = 0; environ[i] != NULL; i++) {

    /* find the '=' in NAME=VALUE */
    char *eq = strchr(environ[i], '=');

    if (eq) {
      /* push name */
      lua_pushlstring(L, environ[i], eq -environ[i]);

      /* push value */
      lua_pushstring(L, eq + 1);

      /* table[name] = value */
      lua_settable(L, -3);
    }
  }

  /* result is the table */
  return 1;
}
```

## *eval* and Environments

A primary characteristic of a dynamic language is the presence of an *eval* construction, which allows the execution of code built at runtime. As we discussed, an *eval* function is also a basic element in an API for a scripting language. In particular, *eval* is the basic means for a host to run scripts.

Lua does not directly offer an *eval* function. Instead, it offers a *load* function. (The code in Figure 2 uses the luaL_loadstring function, which

is a variant of *load*.) This function does not execute a piece of code; instead, it produces a Lua function that, when called, executes the given piece of code.

Of course, it is easy to convert *eval* into *load* and vice versa. Despite this equivalence, we think *load* has some advantages over `eval`. Conceptually, *load* maps the program text to a value in the language instead of mapping it to an action. An *eval* function is usually the most complex function in an API. By separating "compilation" from execution, it becomes a little simpler; in particular, unlike *eval, load* never has side effects.

The separation between compilation and execution also avoids a combinatorial problem. Lua has three different load functions, depending on the source: one for loading strings, one for loading files, and one for loading data read by a given reader function. (The former two functions are implemented on top of the latter.)

Because there are two ways to call functions (protected and unprotected), we would need six different `eval` functions to cover all possibilities.

Error handling is also simpler, as static and dynamic errors occur separately. Finally, *load* ensures that all Lua code is always inside some function, which gives more regularity to the language.

Closely related to the `eval` function is the concept of environment. Every Turing-complete language can interpret itself; this is a hallmark of Turing machines. What makes `eval` special is that it executes dynamic code in the same environment as the program that is using it. In other words, an `eval` construction offers some level of reflection. For example, it is not too difficult to write a C interpreter in C. But faced with a statement such as $x$=1, this interpreter has no way of accessing variable $x$ in the program, if there is one. (Some non-ANSI facilities, such as those related to dynamic-linking libraries, allow a C program to find the address of a given global symbol, but the program still cannot find anything about its type.)

An environment in Lua is simply a table. Lua offers only two kinds of variables: local variables and table fields. Syntactically, Lua also offers global variables: any name not bound to a local declaration is considered global. Semantically, these unbound names refer to fields in a particular table associated with the enclosing function; this table is called the *environment* of that function. In a typical program, most (or all) functions share a single environment table, which then plays the role of a global environment.

Global variables are easily accessible through the API. Because they are table fields, they can be accessed through the regular API to manipulate tables. For example, function `lua_setglobal`, which appears in the bare-bones Lua application code shown earlier, is actually a simple macro written on top of table-manipulation primitives.

Local variables, on the other hand, follow strict lexical-scoping rules, so they do not take part in the API at all. Because C code cannot be lexically nested inside Lua code, C code cannot access local variables in Lua (except through some debug facilities). This is practically the only mechanism in Lua that cannot be emulated through the API.

There are several reasons for this exception. Lexical scoping is an old and powerful concept that should follow the standard behavior. Moreover, because local variables cannot be accessed from outside their scopes, lexical scoping offers programmers a foundation for access control and encapsulation. For example, any file of Lua code can declare local variables that are visible only inside the file. Finally, the static nature of local variables allows the compiler to place all local variables in registers in the register-based virtual machine of Lua.[5]

## Conclusion

We have argued that providing an API to the outside world is not a detail in the implementation of a scripting language, but instead is a decision that may affect the entire language. We have shown how the design of Lua was affected by its API and vice versa.

The design of any programming language involves many such trade-offs. Some language attributes, such as simplicity, favor embeddability, while others, such as static verification, do not. The design of Lua involves several trade-offs around embeddability. The support for modules is a typical example. Lua supports modules with a minimum of extra mechanisms, favoring simplicity and embeddability at the expense of some facilities such as unqualified imports. Another example is the support for lexical scoping. Here we chose better static verification to the detriment of its embeddability. We are happy with the balance of trade-offs in Lua, but it was a learning experience for us to pass through the eye of that needle. C

**Related articles on queue.acm.org**

**Purpose-Built Languages**
*Mike Shapiro*
http://queue.acm.org/detail.cfm?id=1508217

**A Conversation with Will Harvey**
*Chris Dibona*
http://queue.acm.org/detail.cfm?id=971586

**People in Our Software**
*John Richards, Jim Christensen*
http://queue.acm.org/detail.cfm?id=971596

**References**

1. de Moura, A., Ierusalimschy, R. Revisiting coroutines. *ACM Trans. Programming Languages and Systems 31*, 2 (2009), 6.1–6.31.
2. DeLoura, M. The engine survey: general results. *Gamasutra*; http://www.gamasutra.com/blogs/MarkDeLoura/20090302/581/The_Engine_Survey_General_results.php.
3. Ierusalimschy, R. *Programming in Lua, 2nd Ed.* Lua.org, Rio de Janeiro, Brazil, 2006.
4. Ierusalimschy, R., de Figueiredo, L. H., Celes, W. Lua—An extensible extension language. *Software: Practice and Experience 26*, 6 (1996), 635–652.
5. Ierusalimschy, R., de Figueiredo, L. H., Celes, W. The implementation of Lua 5.0. *Journal of Universal Computer Science 11*, 7 (2005): 1159–1176.
6. Ierusalimschy, R., de Figueiredo, L. H., Celes, W. The evolution of Lua. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages* (San Diego, CA, June 2007).
7. Ousterhout, J.K. Scripting: Higher-level programming for the 21st century. *IEEE Computer 31*, 3 (1998), 23–30.
8. Python Software Foundation. Extending and embedding the Python interpreter, Release 2.7 (Apr. 2011); http://docs.python.org/extending/.

**Roberto Ierusalimschy** is an associate professor of computer science at PUC-Rio (Pontifical Catholic University of Rio de Janeiro), where he works on programming-language design and implementation. He is the leading architect of the Lua programming language and the author of *Programming in Lua* (now in its second edition).

**Luiz Henrique de Figueiredo** is a full researcher and a member of the Vision and Graphics Laboratory at the National Institute for Pure and Applied Mathematics in Rio de Janeiro. He is also a consultant for geometric modeling and software tools at Tecgraf, the Computer Graphics Technology Group of PUC-Rio, where he helped create Lua.

**Waldemar Celes** is an assistant professor in the computer science department at Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and a former postdoctoral associate at the Program of Computer Graphics, Cornell University. He is part of the computer graphics technology group of PUC-Rio, where he coordinates the visualization group. He is also one of the authors of the Lua programming language.