The Evolution of an Extension Language: A History of Lua

Roberto Ierusalimschy Luiz Henrique de Figueiredo Waldemar Celes

TeCGraf, Department of Computer Science, PUC-Rio

lua@tecgraf.puc-rio.br

Abstract

Since its inception, in 1993, the Lua programming language has gone far beyond our most optimistic expectations. In this paper, we describe the trajectory of Lua, from its creation as an in-house language for two specific projects, until Lua 4.0, released in November 2000. We discuss the evolution of some of its concepts and the main landmarks in its implementation.

1 Introduction

There is an old joke that says that "a camel is a horse designed by a committee". Among programming-language people, this joke is almost as popular as the legend about programming languages designed by committees. This legend is supported by languages like Algol 68, PL/I, and Ada, all designed by committees, which did not fulfill the expectations of their sponsors.

However, apart from committees, there is an alternative theory for the partial failure of those languages: All of them were born to be big. Each of them followed a top-down design process, where the language was fully specified before any programmer could try it, even before any compiler was built.

Most successful languages, on the other hand, are raised rather than designed. They follow a bottom-up process, starting as a small language, usually with modest goals. As people start using the language, design flaws surface, new features are added (or, eventually, removed), controversial points are clarified (or, eventually, obscured). The way languages evolve, therefore, is an important topic of study in programming languages. For instance, SIGPLAN has already sponsored two conferences on history of programming languages [3, 31].

In this paper, we report the history of the Lua programming language. Since its inception, as an in-house language for two specific projects, Lua has gone far beyond our most optimistic expectations. We think that the main reasons for this success lie in our original design decisions: keep the language simple and small; keep the implementation simple, small, fast, portable, and free.

Lua was designed (or, more exactly, raised) by a committee; a rather small one, with only three members, but a committee. With hindsight, we consider that being raised by a small committee was very positive for the language. We only include a new feature when we reach unanimity; otherwise, it is left for the future. It is much easier to add features later than to remove them. This development process has been essential to keep the language simple, and simplicity is our most important asset. Most other qualities of Lua — speed, small size, and portability — derive from its simplicity.

Since its first version Lua has had “real” users, that is, users others than ourselves. They always gave important contributions to the language, through suggestions, complaints, use reports, and questions. Again, our small committee plays an important role. Its structure gives us enough inertia, so that we can listen to the users without following all their suggestions.

We organized the rest of this paper chronologically. We start with our previous experiences that led to the creation of Lua, in 1993, and proceed through eight years of discussions, decisions, work, and fun.

2 The Beginning

Our first experience at TeCGraf with a language designed in-house arose in a data-entry application. The engineers at PETROBRAS (the Brazilian oil company) needed to prepare input data files for simulators several times a day. This process was boring and error-prone because the simulation programs were legacy code that needed strictly formatted input files — typically bare columns of numbers, with no indication of what each number meant. Of course, each number had a specific meaning, which the engineers could grab at a glance, once they saw a *diagram* of the particular simulation. TeCGraf was asked by PETROBRAS to create several graphical front-ends for this kind of data entry. The numbers could then be input interactively, just by clicking at the relevant parts of the diagram — a much easier and meaningful task than editing columns of numbers. Moreover, it opened the opportunity to add data validation and also to compute derived quantities from the input data. reducing the amount of data needed from the user, and increasing the reliability of the whole process.

To simplify the development of these front-ends at TeCGraf, we decided to code them all in a uniform way, and so we designed a simple declarative language to describe each data entry task [12]. Here is a section of a typical program in this language, which we called DEL (data entry language):

```
:e      gasket              "gasket properties"
mat     s                    # material
m       f       0            # factor m
y       f       0            # settlement stress
t       i       1            # facing type

:p
gasket.m>30
gasket.m<3000
gasket.y>335.8
gasket.y<2576.8
```

The statement `:e` defines an entity (called `gasket`, in the example), which has some fields with default values. The statement `:p` defines some restrictions on the values of `gasket`, and so implements data validation. DEL had also statements to specify how data was input and output.

An entity in DEL is essentially a structure or record in conventional programming languages. The difference is that its name also appears in a graphical metafile, which contains the associated diagram over which the engineer does the data entry, as described above.

This simple language proved to be a success, both in TeCGraf, because it simplified the development, and with the users, because it was simple to tailor data entry applications. Soon users began to demand more power from DEL, such as boolean expressions for controlling whether an entity was active for input or not, and DEL became heavier. When they began to ask for conditional control and loops, it was clear that we needed a true programming language.

At about the same time we started working in another project for PETROBRAS, called PGM, a configurable report generator for lithology profiles. As the name suggests, the reports generated by this program are highly configurable: the user can create and position the tracks, choose colors, fonts, and texts; each track may have a grid, which also has its set of options (log/linear, vertical and horizontal ticks, etc.); each curve has its own scale, which has to be changed automatically in case of overflow; and so on.

All this configuration was to be done by the end users, typically geologists or engineers, and the program should run on small machines, such as a PC running MS-DOS. We decided that the best way to configure this application was through a specialized description language, which we called *Sol*: an acronym for *Simple Object Language*, which also means *sun* in Portuguese.

Because the report generator had many different objects, each one with many different attributes, we did not fix those objects and attributes in the language. Instead, the language allowed type declarations. The main task of the interpreter was to read a description, to check whether the given objects and attributes were correctly typed, and then to present the information to the main program. To allow this communication between the main program and the interpreter, the latter was implemented as a C library, linked to the main program. Thus, the main program could access all configuration information through an API in this library. Moreover, the program could register a callback function for each type, so that the interpreter would call this function whenever it created an object of the given type.

The following chunk shows a typical piece of code in Sol:

```
-- defines a type ‘track’, with numeric attributes ‘x’ and ‘y’,
-- plus an untyped attribute ‘z’. ‘y’ and ‘z’ have default values.
type @track { x:number,y:number= 23, z=0}

-- defines a type ‘line’, with attributes ‘t’ (a track),
-- and ‘z’, a list of numbers.
-- ‘t’ has as default value a track with x=8, y=23, and z=0.
type @line { t:@track=@track{x=8},z:number*}

-- creates an object ‘t1’, of type ‘track’
t1 = @track { y = 9, x = 10, z="hi!"}

-- creates a line ‘l’, with t=@track{x=9, y=10},
-- and z=[2,3,4] (a list)
l = @line { t= @track{x=t1.y, y=t1.x}, z=[2,3,4] }
```

The syntax of Sol was strongly influenced by BiBTeX [21] and UIL (User Interface Language), a language for describing user interfaces in Motif [24].

In March 1993 we finished a first implementation of the Sol language, but we never delivered it. By mid-1993, we realized that both DEL and Sol could be combined into a single, more powerful language. The program for visualizing lithology profiles would soon require support for procedural programming to allow the creation of more sophisticated layouts. On the other hand, the data-entry programs also needed descriptive facilities for programming its user interface.

So, we decided that we needed a real programming language, with assignment, control structures, sub-routines, and the like. The language should also offer data-description facilities, such as those offered by Sol. Moreover, because many potential users of the language were not professional programmers, the language should avoid cryptic syntax (and semantics). Finally, the implementation of the new language should be highly portable.

The portability requirement turned out to be one of its main strengths: Those two applications should be fully portable, and so should the language. The state-owned PETROBRAS could not choose specific hardware because it could only purchase equipment under very strict rules for spending public money. PETROBRAS thus had a very diverse collection of computers, and so the software developed at TeCGraf for PETROBRAS should run on every machine they had, which included PC DOS, Windows (3.1, at that time), Macintosh, and all flavors of Unix.

At that point, we could have adopted an existing language, instead of creating a new one. The main contenders were Tcl [25] and, far behind, Forth [26] and Perl [30]. Perl is not an extension language. In 1993, Tcl and Perl ran only on Unix platforms. All three have highly cryptic syntax. And none of them offers good support for data description. So, we started working on a new language.

Soon we realized that, for our purposes, the language did not need type declarations. Instead, we could use the language itself to write type-checking routines, provided that the language offered basic reflexive facilities (such as run-time type information). An assignment like

```
t1 = @track {y = 9, x = 10, z="hi!"}
```

which was valid in Sol, would be also valid in the new language, but with a different meaning: It creates an object (that is, an associative table) with the given fields, and then calls function `track` to validate the object (and, eventually, to provide default values).

Because the new language was a modified version of Sol (*sun*), a friend at TeCGraf suggested the name Lua (*moon*, in Portuguese), and Lua was born.

Lua inherited from Sol the syntax for record and list constructions, but it unified their implementation using associative tables: Records use strings (the field names) as indices; lists use integers. Apart from these data description facilities, Lua had no new concepts; we wanted just a light generic language. So, we settled for a small set of control structures, with syntax borrowed from Modula (`while`, `if`, and `repeat until`). From CLU we took multiple assignment and multiple returns from function calls (a much cleaner concept than in-out or reference parameters). From C++ we took the neat idea of allowing a local variable to be declared only where we need it.

One of the few (rather small) innovations was the syntax for string concatenation. Because the language allows coercion of strings to numbers, a `+` signal would be ambiguous; so, we created the syntax `..` (two dots) for that operation.

A polemic point was about the use of semicolons. We thought that requiring semicolons could be a little confusing for engineers with a FORTRAN background, but not allowing them could confuse those with a C or Pascal background. At the end, we settled for optional semicolons (a typical committee solution).

Initially, Lua language had seven types: numbers (implemented as floats), strings, (associative) tables, nil (a type with a unique value also called nil), userdata (a generic C pointer to represent C structures inside Lua), Lua functions, and C functions. (After eight years of continuous evolution, the only change in Lua types was the unification of Lua functions and C functions into a single *function* type.) To keep the language small, we did not include a boolean type. Like in Lisp, nil represents false, and any other value represents true. This is one of the few economies that we sometimes regret today.

Lua also inherited from Sol the concept of being implemented as a library. The implementation followed a tenet now supported by Extreme Programming: "the simplest thing that could possibly work" [1]. We used `lex` for the scanner and `yacc` for the parser. The parser translated the program to a bytecode form, which was then executed by a simple stack-based interpreter. The language had a very small pre-defined library, as it was easy to add new functions in C.

Despite this simple implementation — or possibly because of it — Lua surpassed our expectations. Both PGM and the Data Entry (ED) projects used Lua successfully [16] (PGM is still in use today). Soon, other projects inside TeCGraf began to use Lua, too.

3 The First Years (1994–1996)

New users create new demands. Not surprisingly, one of the first demands on Lua was for better performance. The use of Lua for data description posed unusual challenges for typical scripting languages.

As soon as we started using Lua, we identified its potential use as the support language for graphics metafiles. The data description facilities of Lua allow its use as a graphics format. Compared with other programmable metafiles, Lua metafiles have the advantage of being based on a truly procedural language. The VRML format, for instance, uses Javascript to model procedural objects, resulting in a heterogeneous (and consequently unclean) code [2]. With Lua, incorporating procedural objects in a scene description is natural. Procedural code fragments can be combined with declarative statements to model complex objects, while preserving clarity.

The data-entry program (ED) was the first one to use Lua for its graphics metafiles. It was not uncommon for a diagram to have several thousand parts, which were described with a Lua constructor with several thousand items, in a file with hundreds Kbytes. That meant that Lua had to cope with huge programs and huge expressions, from a programming-language perspective. And, because Lua pre-compiled such programs on the fly (a "just-in-time compiler"), it also meant that the Lua compiler had to be very fast. The first victim of this quest for performance was `lex`. Replacing the scanner generated by `lex` with a hand-written one almost doubled the speed of the Lua compiler.

We also created new opcodes for constructors. The original code for a list constructor like

```
@[30, 40, 50]
```

was something like

```
CREATETABLE
PUSHNUMBER 1                # index
PUSHNUMBER 30               # value
SETTABLE
PUSHNUMBER 2                # index
PUSHNUMBER 40               # value
SETTABLE
PUSHNUMBER 3                # index
PUSHNUMBER 50               # value
SETTABLE
```

With the new scheme, the code looked like this:

```
CREATETABLE
PUSHNUMBER 30              # value
PUSHNUMBER 40              # value
PUSHNUMBER 50              # value
SETTABLE 1 3               # set elements from index 1 to 3
```

For a long constructor, it was impossible to push all its elements into the stack before storing them; so the code generator issued a SETTABLE instruction from time to time, to flush the stack.

(Since then, we have always tried to improve compile time. Today, Lua compiles a program with 30000 assignments six times faster than Perl, eight times faster than Python.)

We released a new version of Lua, with those optimizations, in July 1994, with the name Lua 1.1 [19]. This version was made available for download by ftp. The previous version, which was never publicly released, then got the name Lua 1.0. Around that time, we also published the first paper describing Lua [10].

Lua 1.1 had a restrictive user license. It was free for academic uses, but not for commercial uses. (Despite the license, it was always open source.) But that license did not work. Most competitors, such as Perl and Tcl, were free. Moreover, the commercial restriction discourages even academic uses, as several academic projects plan to eventually go to the market. So we released the next version of the language, Lua 2.1, as free software.

3.1 Lua version 2

Lua 2.1 (released in February 1995), brought many important changes. One of them was not in the language itself, but in the process of language development: We decided that we should always try to improve the language, even at the cost of small incompatibilities with previous versions.

In version 2.1 we actually introduced big incompatibilities with version 1.1 (but we provided some tools to help in the conversion). We dropped the `@` from constructors, and unified the use of curly brackets both for records and for lists. Dropping the `@` is a trivial change, but it actually changed the feel of the language, not merely its looks.

More importantly, we simplified the semantics for constructors. In Lua 1.1, the expression `@track{x=1, y=10}` had a special meaning. In Lua 2.1, the expression `track{x=1, y=10}` is just syntactic sugar for `track({x=1, y=10})`, that is, it creates a new table and passes it as the sole argument to function `track`.

Since the beginning, we designed Lua as an extension language, in the sense that C programs can register their own functions to be called from Lua transparently. In this way, it is easy to extend Lua with domain-specific primitives, so that the end user uses a language tailored to her needs.

In version 2.1, we introduced the notion of *fallbacks*: User-defined functions that are called whenever Lua does not know how to proceed. Lua then became a language that could be extended in two ways: by extending its set of "primitive" functions and by extending its semantics with fallbacks. That is why we now call Lua an *extensible* extension language.

We defined fallbacks for arithmetic, comparison, string concatenation, table access, etc. When set by the user, the corresponding fallback function is called whenever the operands to these operations are not of the required type. For instance, whenever two values are added and one of them is not numerical, a fallback is called and its return value is used as the result of the addition.

Of special interest — and actually the main reason for introducing fallbacks — are the fallbacks for table access: in the statement `x=a[i]`, if `a[i]` is nil, then a fallback is called (if set) and its return value used as the value of `a[i]`. This simple new feature allowed the programmer to implement different semantics for table access. In particular, one could implement several kinds of *inheritance*, the simplest one being single inheritance by delegation:

```
function Index (a,i)
  if i == "parent" then       -- to avoid loop
    return nil
  end
  local p = a.parent
  if type(p) == "table" then
    return p[i]               -- may trigger Index again
  else
    return nil
  end
end

setfallback("index", Index)
```

This code follows a chain of "parents" upwards, until a table has the required field or the chain ends. With the "index" fallback set as above, the code below prints `red` even though `b` does not have a `color` field:

```
a=Window{x=100, y=200, color="red"}
b=Window{x=300, y=400, parent=a}
print(b.color)
```

There is nothing magical or "hard-coded" about delegation through a "parent" field. This is a programmer choice. She could use a different name for the "parent" field, or implement more complicated multiple inheritance by allowing the "parent" field to be itself a table of parents that are tried in order, or anything else.

Different fallbacks were called for the expression `a[i]` when `a` is not a table. There was a "gettable" fallback, triggered to get the value of `a[i]`, as in `x=a[i]`; and a "settable" fallback, triggered to set the value of `a[i]`, as in `a[i]=x`.

There are many possibilities for exploiting these table fallbacks; *cross-language inheritance* is a very powerful one: When `a` is a userdata value (a pointer in the host C program), table fallbacks can give the programmer transparent access to values in data structures residing in the host program.

Our decision not to hard-code any of those possible behaviors led to one of the main design concepts of Lua: *meta-mechanisms*. Instead of littering the language with lots of features, we provided ways so that the user could program the features herself, in the way she wants them, and *only* for those features she needs.

The fallback meta-mechanism allowed Lua to support object-oriented programming, in the sense that (several kinds of) inheritance (and also operator overloading) could be implemented. We even added a piece of syntactical sugar for defining and using "methods": functions can be defined as `a:f(x,y,z)` and a hidden parameter called `self` is added to `a.f`, in the sense that a call to `a:f(10,20,30)` is equivalent to `a.f(a,10,20,30)`.

In May 1996 we released Lua 2.4. A main feature of this new version was an external compiler, called `luac`. This program pre-compiles Lua code and saves bytecode and string tables to a binary file. The format of this file was chosen to be easily loaded and portable across different platforms. With `luac`, programs can avoid parsing and code generation at run-time, which can be costly, specially for large, static programs such as graphical metafiles.

Our first paper about Lua [10] already anticipated the possibility of an external compiler, but we only needed it after Lua became widely used at TeCGraf and large graphical metafiles were written in Lua, as the output of graphical editors.

Besides faster loading, `luac` also allows off-line syntax checking and the protection of source code from user changes. However, pre-compiling does not imply faster execution because Lua chunks are always compiled into bytecodes before being executed — `luac` simply allows those bytecodes to be saved in a file for later execution.

`luac` is implemented in "client mode", that is, it uses the modules which implement Lua simply as a (polite) client, even though it does include the private header files to have access to the internal data structures that need to be saved. One advantage of this policy is that it helped to structure the implementation of Lua's core into clearly separated modules. In particular, it is now easy to remove the parsing modules (lexer, parser, and code generator), which represent 40% of the core code in Lua 4.0, leaving just the tiny module that loads pre-compiled chunks. This can be useful for tiny implementations of Lua to be embedded in small devices such as mobile devices or robots[1].

4 International Exposure (1996–2000)

In June 1996 we published an academic paper about Lua in *Software: Practice & Experience* [18]. In December 1996, the magazine *Dr. Dobb's* featured an article about Lua [11]. These publications, aimed at different communities, started to make Lua internationally known.

Soon after the article in Dr. Dobb's, we received several messages about Lua. One of the first messages was the following:

```
From: Bret Mogilefsky <mogul@lucasarts.com>
To: "'lua@icad.puc-rio.br'" <lua@icad.puc-rio.br>
Subject: LUA rocks!  Question, too.
Date: Thu, 9 Jan 1997 13:21:41 -0800

Hi there...

After reading the Dr. Dobbs article on Lua I was very eager to check it
out, and so far it has exceeded my expectations in every way!  It's
elegance and simplicity astound me.  Congratulations on developing such
a well-thought out language.

Some background: I am working on an adventure game for the LucasArts
Entertainment Co., and I want to try replacing our older adventure game
scripting language, SCUMM, with Lua.

[...]
```

It turned out that Bret Mogilefsky was the lead programmer on Grim Fandango, the main adventure game LucasArts released in 1997. In another message he told us that "A TREMENDOUS amount of this game is written in Lua" (his emphasis). This first use of Lua in a game attracted the attention of many game developers around the world to the language. Soon after,

[1] Crazy Ivan, a robot that won RoboCup in 2000 and 2001 in Denmark, has a "brain" implemented in Lua.

Lua started to appear frequently in game newsgroups, such as `rec.games.programmer` and `comp.ai.games`.

Due to its small size, good performance, portability and ease of integration, Lua has gained great popularity for extending games. Nowadays, several game companies use Lua (e.g., LucasArts, BioWare, Slingshot Game Technology, and Loewen Entertainment) and knowledge of Lua is a marketable skill in the game industry. We estimate that half of the Lua users are in some way involved in game programming, but it is hard to be more specific because there is a lot of secrecy in the game industry. For instance, Bret Mogilefsky adapted Lua for Grim Fandango, but the details were of course proprietary.

There are several benefits from embedding a scripting language into a game. The scripting language can be used for defining sprite and object physics, for managing object AI and character control, and for handling input device events. For instance, the engine may know nothing about such things as “damage”, “speed”, “weapons”, etc. The choice of a simple language can also give game designers access to programmable tools. This is crucial for game development, because the designers can experiment their creations. Scripting languages also allow rapid prototyping and facilitates the implementation of debugger tools.

More recently, in 2000, LucasArts released another game using Lua: *Escape from Monkey Island*, which was the fourth in the Monkey Island series of adventures. In this game, as a homage to Lua, they renamed a bar inside the game from *SCUMM* (the language they used previously) to *Lua* Bar.

Besides its wide use in computer games (e.g., Grim Fandango, Baldur’s Gate, MDK2, Escape from Monkey Island), Lua has been used around the world in many different fields.

One of the first uses of Lua outside PUC-Rio was at the Smithsonian Astrophysical Observatory. They designed a generalized aperture program to simulate the effects on the incident photon stream of physical obstructions, and used Lua to model the geometry and the interactions of the incident photon stream with the apertures [23]. This program was part of the efforts to support the AXAF program (the Advanced X-ray Astrophysics Facility) — the third of NASA’s four Great Space Observatories.

Performance Technologies used Lua to implement the command-line interface of CPC4400, a hot-swappable Ethernet switch. By exposing Lua as the scripting language of the CPC4400, the user is able to associate events (such as link status, detection of topology changes and RMON alarms) with Lua scripts.

Tollgrade Communications used Lua in its next-generation telephony network testing products, *DigiTest*. Lua was used for user interfaces, automated test scripts, and results analysis.

Lua is also used at the InCor Heart Institute (Instituto do Coração, São Paulo), in Brazil; CEPEL (the Research Center of ELETROBRAS, the state electricity company), also in Brazil; the Weierstrass Institute, in Berlin; the Technische Universität Berlin; and many other places.

In 1998, Cameron Laird and Kathryn Soraiz, in their column about scripting languages in SunWorld magazine, estimated that “there might be only a few tens of thousands of Lua programmers in the world” [20]. What they considered “a small user base” was, for us, a strong sign of the growing popularity of the language.

4.1 Lua version 3

Lua 3.0 (July 1997) replaced fallbacks with the more powerful concept of *tag methods*. Fallbacks were global by nature: User functions were called every time an event occurred, and there was only one function for each event. This made it hard to combine Lua modules that had different notions of inheritance, for instance. Although it was possible to chain fallbacks, chaining was slow and error-prone, and in practice nobody did it.

Since Lua 3.0, the programmer can create *tags*, and associate tables and userdata to tags. Tag methods are essentially fallbacks that are selected according to the tag of the operator. With tags and tag methods, different tables (and userdata) may have different fallbacks for their operations.

The tag concept essentially provides Lua with user-defined types. In other words, a tag is simply a number that represents a new type. When you associate a table with a specific tag, you are actually defining a new type for that table: The type (or the tag) of the table specifies how it implements its operators.

When we introduced fallbacks, most of them described the behavior of Lua for otherwise erroneous events, such as indexing a non-table value or calling a non-function value. So, we thought about fallbacks as an exception-handling mechanism. With the introduction of user-defined tags, fallbacks (now called tag methods) became mainly a mechanism for describing the behavior of new types, even though we still can use them to extend the behavior of basic types.

Despite this new status, for a long time we still perceived tag methods as exception-handling mechanisms, and we did not connect tags with types. Only recently we realized the full significance of user-defined tags and tag methods as a mechanism for creating user-defined types. Lua 4.1 will seal this recognition by allowing the user to provide *names* for these new types (currently, only the basic types have names).

Lua 3.0 also brought support for conditional compilation, in the form of a C-like preprocessor. Like any language feature, it was too easy to add it (although it did complicate the lexer) and soon programmers began to use it (programmers will use *any* language feature). Once a feature begins to be used, demand for more power comes right behind it. One of the most frequent requests was the addition of macro expansion, but no clean proposal emerged from the long discussions in the mailing list and among ourselves. Every such proposal would imply a large change in the lexer and parser, and the benefits were not clear. So, the preprocessor remained static from Lua 3.0 until version 3.2 (two years).

In the end we decided that the preprocessor was causing more harm than good, making the code bulkier and inviting endless discussions from users, and we removed it in Lua 4.0. We feel that Lua is now cleaner without the preprocessor. Over the years, we have striven to make Lua simpler and have removed dark corners of the language that we once regarded as features, but which few programmers really used and which were later recognized as misfeatures.

4.2 Lua version 4

Until version 3.2, only one Lua "state" could be active at a time. We did have an API function to change the state, but it was a little awkward to use. For simplicity, when we designed the API, we did not include an explicit state parameter in the functions — there was single, global state. In retrospect, that was a mistake. By the time Lua 3.2 was out, it was clear that many applications would be simpler if they could run multiple Lua states in a convenient way. For instance, we had to make a special version of Lua 3.2 to be included in CGILua, an extension of web browsers for dynamic page serving and CGI programming in Lua. Earlier, LucasArts did something similar for Lua 3.1.

When we were discussing our plans for Lua 3.3, an API with explicit states was our highest priority. However, this raised compatibility questions. In the end, because there would have to be several incompatibilities, we decided to rewrite the API and the next version became Lua 4.0. The API now not only includes explicit states, but it is also easier to use and more efficient. We feared that the transition to the new API would be a little traumatic, since it was the first time we really changed the API since Lua 1.1. We did have a few complaints in the mailing list, but on the whole the change was not at all traumatic. Most Lua programmers have no contact with the API, many use it only through automatic tools, and many considered that the benefits outperformed the transition costs.

We released Lua 4.0 in November 2000. Besides the new API, this version brought many other small enhancements, among them the `for` loop.

Everyone that works with programming languages knows how easy it is for people to start "religious wars" about the subject. An interesting characteristic of those wars is that, usually, the more mundane the subject, the hotter the discussion. For instance, people get much more excited discussing semicolons than discussing higher-order functions. Of course, one reason for that is that many more people have opinions about the former than the latter. But another, more important reason is that mundane details have a strong impact in how comfortable people feel with the language. It is no use to create a marvelous, well-thought tool, if it does not have a good grip — no one will use it.

Since version 1.1, a `for` statement was in the wish-list of most Lua users. The most common complaint was that people forget to write the increment at the end of a while loop, thus leading to infinite loops.

It did not take too long for us to agree. But, while we all agreed about the need of a `for` loop, we could not agree about any particular construction. We considered a Pascal-like (or Modula-like) construction too restrictive, as it does not contemplate iterations over the elements of a table or over the lines of a file. Moreover, turning the identifier `to` into a reserved word would be an unacceptable incompatibility. On the other hand, a `for` loop in the C tradition did not fit in Lua.

With the introduction of closures and anonymous functions, in version 3.1 (July 98), we decided to use higher-order functions for iterations. (In fact, the need for iterators was one of the main reasons for introducing anonymous functions in Lua.) Lua 3.1 came out with two pre-defined functions for iteration:

```
foreach(table, f)
foreachi(table, f)
```

The `foreach` function applies `f` over all pairs key-value in the given table, in no specific order. The `foreachi` function is similar, but it sees the table as a list (or an array): it traverses only the elements with numeric indices, and ensure they are traversed in ascending order. Although we provided functions only for these two particular traversals, it was all to easy to create new iterators.

More than two years after the introduction of these iteration functions we realized that, although it was easy to create new iterators, almost nobody did. The first reason is that most programmers do not feel comfortable with anonymous functions and higher order functions, mainly inside a procedural language. But the second and, in our view, most important reason was that almost nobody *needed* other iterators. That meant that, for many years, we had been trying to achieve an orthogonality that no real user really cared about. With this understanding, we quickly designed a `for` loop, with two formats, one for a numeric loop, the other to traverse tables.

The `for` statement was one of the most successful changes in the language since its first version. First, it really covers most common loops; for a really generic loop, there is the `while` loop. Second, because of its rigid format, it is easy to create specific opcodes to implement the loop, so that a numeric `for` loop with an empty body runs more than twice as fast as the equivalent `while` construction.

5 Conclusions

Currently, Lua has a well-established user base. Lua has an active discussion list, with almost 500 people from more than 30 different countries. Its web site (`www.lua.org`) receives around 500 visits per day, from 50 countries. Its uses range from adventure games to web-servers to telephony-network testing to Ethernet switches.

Several ftp sites offer the original Lua source code, and several other sites distribute versions for specific platforms, such as DLLs for Windows, SIS for EPOC, RPM for Linux, binaries for RISC OS, etc. Moreover, several magazines have distributed Lua in supplementary CDs (e.g. Dr. Dobb's, Linux Magazine France, and Japan's C Magazine).

As a language, Lua main contributions are a consequence of the decision to provide meta-mechanisms instead of features. As a product, Lua success comes from its simplicity, small size, and from the portability of the implementation, which allows Lua to be used in many different platforms, including small devices, such as palmtops, handhelds, specialized boards, and robots. Cameron Laird and Kathryn Soraiz predicted, in 1998, that "the imminent explosion of ubiquitous embedded processing (computers in your car, in your plumbing, and in your kitchen appliances) can only work in favor of Lua" [20]. At that time, we did not pay much attention, but they were right.

Lua also made several academic contributions, through several thesis and papers both about Lua and using Lua as a relevant technology [4–9, 13–15, 17, 22, 27–29].

Success has its costs. Through the evolution of the language, compatibility with previous versions has been increasingly a deterrent to innovation. Nevertheless, we do not allow compatibility to stop progress; it is only one more ingredient (albeit a strong one) in the alchemy of language design.

Finally, to keep a language is much more than to design it. Complete attention to detail is essential in all aspects: language design, implementation, documentation, setting up a community of users and listening to them, while at the same time keeping to the original design decisions.

Acknowledgements

Lua would never be what it is without the help of many people. Everyone in TeCGraf contributed in different forms — using the language, discussing it, disseminating it outside TeCGraf. Special thanks go to Marcelo Gattass, head of TeCGraf, who always encouraged us, and gave us complete freedom over the language and its implementation. Lua was actually the first TeCGraf product to be publicly available in the internet, even before the boom.

Without users Lua would be just yet another language. Users and their uses are the ultimate test for a language. From them we got bug reports, design flaws, new ways of seeing reality. Special thanks go to the members of the discussion list, for their suggestions, complaints, and mainly for bearing with our sometimes autocratic style.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[2] G. Bell, R. Carey, and C. Marrin. The Virtual Reality Modeling Language Specification—Version 2.0. `http://www.vrml.org/VRML2.0/FINAL/`, Aug. 1996. (ISO/IEC CD 14772).

[3] T. J. Bergin and R. G. Gibson, editors. *History of Programming Languages*, volume 2. ACM Press, 1996.

[4] A. Carregal and R. Ierusalimschy. Tche — a visual environment for the Lua language. In *VIII Simpósio Brasileiro de Computação Gráfica*, pages 227–232, São Carlos, 1995.

[5] W. Celes. *Modelagem configurável de subdivisões planares hierárquicas*. PhD thesis, Dep. Informática, PUC-Rio, Rio de Janeiro, Brazil, 1995.

[6] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic component gluing across different componentware systems. In *DOA'99 — International Symposium on Distributed Objects and Applications*, pages 362–371, Edinburgh, Escócia, 1999. IEEE Computer Society.

[7] M. T. M. de Carvalho. *Uma estratégia para o desenvolvimento de aplicações configuráveis em mecânica computacional*. PhD thesis, Dep. Engenharia Civil, PUC-Rio, Rio de Janeiro, Brazil, June 1995.

[8] R. F. de Gusmão Cerqueira. *Um Modelo de Composição Dinâmica entre Sistemas de Componentes de Software*. PhD thesis, Dep. Informática, PUC-Rio, Rio de Janeiro, Brazil, Aug. 2000.

[9] M. J. de Lima, N. Rodriguez, and R. Ierusalimschy. Remote functions as first-class values in a distributed object system. In *IV Simpósio Brasileiro de Linguagens de Programação*, pages 1–14, Recife, May 2000.

[10] L. H. Figueiredo, R. Ierusalimschy, and W. Celes. The design and implementation of a language for extending applications. In *XXI Semish*, pages 273–284, Caxambu, 1994.

[11] L. H. Figueiredo, R. Ierusalimschy, and W. Celes. Lua: An extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, Dec. 1996.

[12] L. H. Figueiredo, C. S. Souza, M. Gattass, and L. C. G. Coelho. Geração de interfaces para captura de dados sobre desenhos. In *Proceedings of SIBGRAPI '92 (Brazilian Symposium on Computer Graphics and Image Processing)*, pages 169–175, 1992.

[13] P. R. Gomes, B. Feijó, R. Cerqueira, and R. Ierusalimschy. Reactivity and pro-activeness in virtual prototyping. In I. Horvath and A. Taleb-Bendiab, editors, *2nd International Symposium on Tools and Methods for Concurrent Engineering*, pages 242–253, Manchester, U.K., Apr. 1998.

[14] T. G. Gorham and R. Ierusalimschy. Um sistema de depuração reflexivo para uma linguagem de extensão. In R. Bigonha, editor, *I Simpósio Brasileiro de Linguagens de Programação*, pages 103–114, Belo Horizonte, Sept. 1996.

[15] A. Hester, R. Borges, and R. Ierusalimschy. Building flexible and extensible Web applications with Lua. *Journal of Universal Computer Science*, 4(9):748–762, 1998. http://medoc.springer.de:8000/.

[16] R. Ierusalimschy, W. Celes, L. H. Figueiredo, and R. de Souza. Lua: Uma linguagem para customização de aplicações. In *VII Simpósio Brasileiro de Engenharia de Software — Caderno de Ferramentas*, page 55, Rio de Janeiro, Brazil, 1993.

[17] R. Ierusalimschy, R. Cerqueira, and N. Rodriguez. Using reflexivity to interface with CORBA. In *IEEE International Conference on Computer Languages (ICCL'98)*, pages 39–46, Chicago, IL, May 1998. IEEE Computer Society.

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

[19] R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Reference manual of the programming language Lua. Monografias em Ciência da Computação 3/94, PUC-Rio, Rio de Janeiro, Brazil, 1994.

[20] C. Laird and K. Soraiz. 1998: Breakthrough year for scripting. *SunWorld*, Aug. 1998. `http://sunsite.icm.edu.pl/sunworldonline/swol-08-1998/swol-08-regex.html`.

[21] L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1986.

[22] M. C. Martins, N. Rodriguez, and R. Ierusalimschy. Dynamic extension of CORBA servers. In *Euro-Par'99 Parallel Processing*, pages 1369–1376, Toulouse, France, 1999. Springer-Verlag. (LNCS 1685).

[23] D. Nguyen, T. Gaetz, D. Jerius, and I. Stern. Modeling AXAF obstructions with the generalized aperture program. In *Astronomical Data Analysis Software and Systems VI*, pages 485–487, Sept. 1996.

[24] Open Software Foundation. *OSF/Motif Programmer's Guide*. Prentice-Hall, Inc., 1991. (ISBN 0-13-640673-4).

[25] J. Ousterhout. Tcl: an embeddable command language. In *Proc. of the Winter 1990 USENIX Conference*. USENIX Association, 1990.

[26] E. D. Rather, C. H. Moore, and D. R. Colburn. The evolution of Forth. In *ACM SIGPLAN History of Programming Languages Conference*, Apr. 1993.

[27] N. Rodriguez and R. Ierusalimschy. Dynamic reconfiguration of CORBA-based applications. In J. Pavelka, G. Tel, and M. Bartošek, editors, *SOFSEM'99: 26th Conference on Current Trends in Theory and Practice of Informatics*, pages 95–111, Milovy, Czech Republic, 1999. Spinger-Verlag. (LNCS 1725).

[28] N. Rodriguez, R. Ierusalimschy, and R. Cerqueira. Dynamic configuration with CORBA components. In *4th International Conference on Configurable Distributed Systems (ICCDS'98)*, pages 27–34, Annapolis, MD, May 1998. IEEE Computer Society.

[29] N. Rodriguez, C. Ururahy, R. Ierusalimschy, and R. Cerqueira. The use of interpreted languages for implementing parallel algorithms on distributed systems. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing — Second International Euro-Par Conference*, pages 597–600, Volume I, Lyon, France, Aug. 1996. Springer-Verlag. (LNCS 1123).

[30] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[31] R. L. Wexelblat, editor. *History of Programming Languages*. Academic Press, 1981.

www.lua.org