CHAPTER SEVEN

Lua

Lua is a very small, self-contained dynamic language created by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes in 1993. Lua's small set of powerful features and easy-to-use C API make the language easy to embed and extend to express domain-specific concepts. Lua is prominent in the world of proprietary software, where games such as Blizzard's *World of Warcraft* and Crytek GmbH's *Crysis*, as well as Adobe's Photoshop Lightroom, use it for scripting and UI work. Its predecessors are Lisp, Scheme, and perhaps AWK; it has design similarities to JavaScript, Icon, and Tcl.

The Power of Scripting

How do you define Lua?

Luiz Henrique de Figueiredo: An embeddable, lightweight, fast, powerful scripting language.

Roberto Ierusalimschy: Unfortunately, more and more people use “scripting language” as a synonym for “dynamic language.” Nowadays even Erlang or Scheme are called scripting languages. That is sad, because we lose the precision to describe a particular class of dynamic languages. Lua is a scripting language in the original meaning of the expression. A language to control other components, usually written in another language.

What should people keep in mind when designing software with Lua?

Luiz: That there probably is a Lua way of doing things. It’s not recommended to try to emulate all practices from other languages. You have to really use the features of your language, although I guess that is true for any language. In the case of Lua, those features are mainly tables for everything and metamethods for elegant solutions. Also coroutines.

Who should use Lua?

Roberto: I think that most applications without a scripting facility could benefit from Lua.

Luiz: The problem is that most designers do not see this need untill much later, when much code has already been written in say C or C++, and they feel that it’s too late now. Application designers should consider scripting from the start. This will give them much more flexibility. It will also give them better perspective about performance, by forcing them to think where the application *needs* raw performance and where it does not matter at all, and so can be delegated to the easier, shorter development cycle of scripting.

From a security point of view, what does Lua offer to the programmer?

Roberto: The core of the Lua interpreter is built as a “freestanding application.” This is a term from ISO C that basically means that the program does not use anything from the environment (no `stdio`, `malloc`, etc.). All those facilities are provided by external libraries. With this architecture, it is very easy to create programs with limited access to external resources. For instance, we can create sandboxes within Lua itself, just erasing from its environment whatever we consider dangerous (e.g., `fileopen`).

Luiz: Lua also offers user-defined debug hooks that can be used to monitor the execution of a Lua program and so, for instance, abort it if it takes too long or uses too much memory.

What are the limits of Lua?

Roberto: I think the main limits of Lua are what I consider the limits of any dynamic language. First, even with the most advanced JIT technology (and Lua has one of the best JITs among dynamic languages), you cannot get the performance of a good static language. Second, several complex programs really can benefit from static analysis (mainly static typing).

Why did you decide to use a garbage collector?

Roberto: Lua has always used a garbage collector, since day one. I would say that, for an interpreted language, a garbage collector can be much more compact and robust than reference counting, not to mention that it does not leave garbage around. Given that an interpreted language usually already has self-described data (values with tags and things like that), a simple mark-and-sweep collector can be really simple, and almost does not affect the rest of the interpreter.

And for an untyped language, reference counting can be very heavy. Without static typing, every single assignment may change counts, and so needs a dynamic check both in the old and in the new value of a variable. Later experiences with reference count in Lua did not improve performance at all.

Are you satisfied with the way Lua manages numbers?

Roberto: In my experience, numbers in computers will always be a source of occasional surprises (as they are outside computers, too!). I consider the use of a double as the single numeric type in Lua a reasonable compromise for Lua. We have considered many other options, but most are too slow, too complex, or too memory-hungry for Lua. Even using double is not a reasonable choice for embedded systems, so we can compile the interpreter with an alternative numerical type, such as `long`.

Why did you choose tables as the unifying data constructor in Lua?

Roberto: From my side, I was inspired by VDM (a formal method mainly for software specification), something I was involved when we started Lua. VDM offers three forms of collections: sets, sequences, and maps. But both sets and sequences are easily expressed as maps, so I had this idea of maps as a unifying constructor. Luiz brought his own reasons, too.

Luiz: Yes, I liked AWK a lot, especially its associative arrays.

What value do programmers derive from first-class functions in Lua?

Roberto: Under different names, from subroutines to methods, "functions" have been a staple of programming languages for more than 50 years, so a good support for functions is an asset in any language. The support that Lua offers allows programmers to use several powerful techniques from the functional-programming world, such as representing data as functions. For instance, a shape may be represented by a function that, given *x* and *y*, tells whether that point lies within the shape. This representation makes trivial operations like union and intersection.

Lua uses functions also in some unconventional ways, and the fact that they are first class simplifies those uses. For instance, every chunk (any piece of code that we feed to the interpreter) is compiled like a function body, so any conventional function definition in Lua is always nested inside an outer function. That means that even trivial Lua programs need first-class functions.

Why did you implement closures?

Roberto: Closures are the kind of construct we always want in Lua: simple, generic, and powerful. Since version 1, Lua has had functions as first-class values, and they proved to be really useful, even for "regular" programmers without previous experience with functional programming, but without closures, the use of first-class functions is somewhat restricted. By the way, the term *closure* refers to an implementation technique, not the feature itself, which is "first-class functions with lexical scoping," but *closure* is certainly shorter. :)

How do you plan to deal with concurrency?

Roberto: We do not believe in multithreading, that is, shared memory with preemption. In the HOPL paper,* we wrote, "We still think that no one can write correct programs in a language where *a*=*a*+1 is not deterministic." We can avoid this problem by removing either preemption or shared memory, and Lua offers support for both approaches.

With coroutines, we have shared memory without preemption, but this is of no use for multicore machines. But multiple "processes" can explore quite effectively those machines. By "process" I mean a C thread with its own Lua state, so that, at the Lua level, there is no memory sharing. In the second edition of *Programming in Lua* [Lua.org], I already presented a prototype of such implementation, and recently we have seen libraries to support this approach (e.g., Lua Lanes and luaproc).

You don't support concurrency, but you did implement an interesting solution for multitasking—namely, asymmetrical coroutines. How do they work?

Roberto: I had some experience with Modula 2 (my wife wrote a full interpreter for M-code during her Master's work), and I always liked the idea of using coroutines as a basis for cooperative concurrency and other control structures. However, symmetrical coroutines, as provided by Modula 2, would not work in Lua.

Luiz: In our HOPL paper we explained all those design decisions in great detail.

Roberto: We ended up with this asymmetrical model. The underlying idea is really simple. We create a coroutine with an explicit call to a `coroutine.create` function, giving a function to be executed as the coroutine body. When we resume the coroutine, it starts running its body and goes until it ends or it yields; a coroutine only yields by explicitly calling the yield function. Then, later, we can resume it again, and it will continue from where it stopped.

The general idea is very similar to Python's generators, but with a key difference: a Lua coroutine can yield inside nested calls, while in Python a generator can only yield from its main function. Thinking about the implementation, this means that a coroutine must

* R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The evolution of Lua," Proceedings of ACM HOPL III (2007).

have an independent stack, just like a thread. What is surprising is how much more powerful these "stackful" coroutines are, compared with "flat" generators. For instance, we can implement one-shot continuations on top of them.

Experience

How do you define success in terms of your work?

Luiz: The success of a language depends on the number of programmers using it and on the success of the applications that use it. We don't really know how many people program in Lua, but there certainly are many successful applications using Lua, including several very successful games. Also, the range of applications that use Lua, from desktop image processing to embedded control of robots, shows that there is a clear niche for Lua. Finally, Lua is the only language created in a developing country to have achieved such global relevance. It is the only such language to have ever been featured in ACM HOPL.

Roberto: This is difficult. I work in several fronts, and at each of them I feel success differently. Overall, I would say that common to most of these definitions is "to be known." It is always a great pleasure to be introduced to someone, or to contact someone, and to be recognized.

Do you have any regrets about the language?

Luiz: I don't really have any regrets. In hindsight we could have done some things earlier if we knew how to do them as we do now!

Roberto: I am not sure I regret something specific, but language design involves several tough decisions. For me, the most difficult decisions are those about ease of use. One of the aims of Lua is to be easy for nonprofessional programmers. I do not fit into this category. So, several decisions about the language are not the ideal ones from my perspective as a user. A typical example is Lua's syntax: many uses of Lua benefit from its verbose syntax, but for my own taste I would rather use a more compact notation.

Did you make mistakes of design or implementation?

Luiz: I don't think that we have made any big mistakes in designing or implementing Lua. We just learned how to evolve a language, which is much more than merely defining its syntax and semantics and implementing it. There are also important social issues, such as creating and supporting a community, with manuals, books, websites, mailing lists, chat rooms, etc. We certainly learned the value of supporting a community and also about the hard work that has to be put into that as well as into designing and coding.

Roberto: Luckily we did not make big mistakes, but I think we made many small ones along the way. But we had the chance to correct them as Lua evolved. Of course this annoyed some users, because of incompatibilities between versions, but now Lua is quite stable.

What do you suggest to become a better programmer?

Luiz: Never be afraid to start over, which of course is much easier said than done. Never underestimate the need for attention to detail. Don't add functionality that you *think* will be useful some time in the future: adding it now may prevent you from adding a much better feature later on, when it's really needed. Finally, always aim for the simpler solution. As simple as possible, but not simpler, as Einstein said.

Roberto: Learn new programming languages, but only from good books! Haskell is a language that all programmers should know. Study computer science: new algorithms, new formalisms (lambda calculus, if you do not know it yet, pi calculus, CSP, etc.). Always try to improve your own code.

What's the biggest problem with computer science and how we teach it?

Roberto: I guess there is no such thing as "computer science" as a well-understood corpus of knowledge. Not that computer science is not science, but what is computer science and what is not (and what is important and what is not) is still too ill defined. Many people in computer science do not have a formal background in computer science.

Luiz: I consider myself as a mathematician interested in the role of computers in mathematics, but of course I do like computers a lot. :)

Roberto: Even among those with a formal background there is no uniformity, we miss a common ground. Many people think Java created monitors, virtual machines, interfaces (as opposed to classes), etc.

Are a lot of CS programs glorified job-training programs?

Roberto: Yes. And many programmers do not even have a CS degree.

Luiz: I don't think so, but I'm not employed as a programmer. On the other hand, I think it would be wrong to require programmers to have CS degrees, or certifications, or anything of that sort. A CS degree is no guarantee that one can program well, and many good programmers don't have a CS degree (perhaps this was true when I started; I'm probably too old now). My point is that a CS degree is no guarantee that one can program well.

Roberto: It is wrong to require most professionals to have degrees, but what I meant was that the "culture" in the area is too weak. There are very few things you can assume people must know. Of course a hirer may demand whatever he wants, but there should not be laws requiring degrees.

What is the role of math in computer science and programming in particular?

Luiz: Well, I'm a mathematician. I see math everywhere. I was attracted to programming probably because it definitely has mathematical qualities: precision, abstraction, elegance. A program is a proof of a complicated theorem that you can continually refine and improve, and it actually does something!

Of course I don't think in those terms at all when programming, but I think that learning math is very important to programming in general. It helps you get into a certain frame of mind. It's much easier to program if you're used to thinking about abstract things that have their own rules.

Roberto: According to Christos H. Papadimitriou, "computer science is the new math." A programmer can only go so far without math. In a broader view, both math and programming share the same key mental discipline: abstraction. They also share a key tool: formal logic. A good programmer uses "math" all the time, establishing code invariants, models for interfaces, etc.

A lot of programming languages are created by mathematicians—maybe that is why programming is difficult!

Roberto: I will leave this question to our mathematician. :)

Luiz: Well, I've said before that programming definitely has mathematical qualities: precision, abstraction, elegance. Designing programming languages feels to me like building a mathematical theory: you provide powerful tools to enable others to do good work. I've always been attracted to programming languages that are small and powerful. There's beauty in having powerful primitives and constructs, just like there is beauty in having powerful definitions and basic theorems.

How do you recognize a good programmer?

Luiz: You just know it. Nowadays, I tend to recognize bad programmers more easily—not because their programs are bad (although they frequently are a complicated, unstable mess), but because you can sense they are not comfortable at programming, as if their own programs were a burden and a mystery to them.

How should debugging be taught?

Luiz: I don't think debugging can be taught, at least not formally, but it can be learned by doing it when you are in a debugging session with someone else, perhaps more experienced than you are. You can then learn debugging strategies from them: how to narrow down the problem, how to make predictions and assess outcomes, what is useless and just adds to noise, etc.

Roberto: Debugging is essentially problem solving. It is an activity where you may have to use all intellectual tools you ever learned. Of course there are some useful tricks (e.g., avoid a debugger if you can, use a memory checker if programming in a low-level language like C), but these tricks are only a small part of debugging. You should learn debugging as you learn to program.

How do you test and debug your code?

Luiz: I try mainly to construct and test it piece by piece. I rarely use a debugger. When I do, it's for C code, never for Lua code. For Lua, a few well-placed `print` statements usually work just fine.

Roberto: I follow a similar approach. When I use a debugger, frequently it's only to do a `where` to find where the code is crashing. For C code, a tool like Valgrind or Purify is essential.

What is the role of comments in the source code?

Roberto: Very small. I usually consider that if something needs comments, it is not well written. For me, a comment is almost a note like "I should try to rewrite this code later." I think clear code is much more readable than commented code.

Luiz: I agree. I stick to comments that say something that the code does not make obvious.

How should a project be documented?

Roberto: Brute force. No amount of tools is a substitute for well-written and well-thought-out documentation.

Luiz: But producing good documentation about the evolution of a project is only possible if we have that in mind from the start. That did not happen with Lua; we never planned Lua to grow so much and be as widely used as it is today. When we were writing the HOPL paper (which took almost two years!), we found it hard to recall how some design decisions had been made. On the other hand, if in the early days we had had meetings with formal records, we would probably have lost some of the spontaneity and missed some of the fun.

What factors do you measure during the evolution of a codebase?

Luiz: I would have to say "simplicity of the implementation." With this comes speed and correctness of the implementation. At the same time, flexibility is also an important point, so that you can change an implementation if needed.

How do available hardware resources affect the mindset of programmers?

Luiz: I'm an old guy. :-) I learned programming on an IBM 370. It took hours between punching cards, submitting it to the queue, and getting the printouts. I have seen all kind of slow machines. I think programmers should be exposed to them, because not everyone in the world has the fastest machines. People programming applications for the masses should try them on slow machines to get a feel for the wider user experience. Of course, they can use the best machines for development: it's not fun having to wait a long time for a compilation to finish. In these days of global Internet, web developers should try slow connections, not the hyperfast ones they have at work. Aiming for an average platform will make your product faster, simpler, and better.

In the case of Lua, the "hardware" is the C compiler. One thing that we learned in implementing Lua is that aiming for portability does pay. Almost from the beginning, we have implemented Lua in very strict ANSI/ISO C (C89). This has allowed Lua to run in special hardware, such as robots, printer firmware, network routers, etc., none of which was ever an actual target for us.

Roberto: One golden principle is that you should always treat hardware resources as limited. Of course they are *always* limited. "Nature abhors a vacuum"; any program tends to expand until it uses all available resources. Moreover, at the same time that resources become cheaper in established platforms, new platforms emerge with severe restrictions. It happened with the microcomputer; it happened with mobile phones; it is happening all the time. If you want to be the first to market, you'd better be prepared to be very conscious about what resources your programs need.

What do the lessons about the invention, further development, and adoption of your language say to people developing computer systems today and in the foreseeable future?

Luiz: I think that one has to keep in mind that not all applications are going run in powerful desktop machines or laptops. Many applications are going to run in constrained devices such as cell phones or even smaller devices. People that design and implement software tools should be especially concerned about this issue, because you can never tell where and how your tool is going to be used. So, design for using minimal resources, and you may be pleasantly surprised to see your tool used in many contexts that you did not have as a primary goal and some that you did not even know existed. This has happened with Lua! And for a good reason; we have an internal joke, which is not really a joke: when we discuss the inclusion of a feature in Lua, we ask ourselves, "OK, but will it run in a microwave oven?"

Language Design

Lua is easy to embed and requires very few resources. How do you design for limited resources of hardware, memory, and software?

Roberto: When we started, we did not have those goals very clear. We just had to meet them to deliver our project. As we evolved, those goals became more clear to us. Now, I guess the main point is to be economic in all aspects, all the time. Whenever someone suggests some new feature, for instance, the first question is how much it will cost.

Have you rejected features because they were too expensive?

Roberto: Almost all features are "too expensive" to what they bring to the language. As an example, even a simple `continue` statement did not pass our criteria.

How much benefit does a feature have to add to be worth its expense?

Roberto: No fixed rules, but a good rule is whether the feature "surprises" us; that is, it is useful for things other than its initial motivation. That remind me of another rule of thumb: how many users would benefit from the feature. Some features are useful for only a small fraction of users, while others are useful to mostly everyone.

Do you have an example of a feature you added that is useful to more people?

Roberto: The `for` loop. We resisted even this, but when it appeared, it changed *all the examples* in the book! Weak tables are also surprisingly useful. Not many people use them, but they should.

,ch07.26816 Page 170 Thursday, February 26, 2009 2:35 PM

You waited seven years after version 1.0 before adding the `for` loop. What made you keep it out? What made you include it?

Roberto: We kept it out because we could not find a format for the `for` loop that was both generic and simple. We included it when we found a good format, using generator functions. Actually, closures were a key ingredient to make generators easy and generic enough to use, because with closures the generator function itself can keep internal state during a loop.

Is that another area of expense: updating code to take advantage of new features and newly discovered best practices?

Roberto: People do not have to use new features.

Do people choose one version of Lua and stick with it throughout the lifetime of the project, never upgrading?

Roberto: I guess most people in games do exactly that, but in other areas I think several projects evolve the Lua version they use. As a counterexample, *World of Warcraft* did change from Lua 5.0 to Lua 5.1! However, keep in mind that Lua now is much more stable than when it was younger.

How do you share development responsibilities—in particular, writing code?

Luiz: The first versions of Lua were coded by Waldemar in 1993. Since around 1995, Roberto has written and maintained the bulk of the code. I'm responsible for a small part of the code: the bytecode dump/undump modules and the standalone compiler, luac. We have always done code revisions and sent suggestions by email to the others about changes to the code, and we have long meetings about new features and their implementation.

Do you get much feedback on the language or the implementation from the users? Do you have a formal mechanism for including user feedback in the language and its revisions?

Roberto: We joke that whatever we do not remember was not that important in the first place. The Lua discussion list is quite active, but some people equate open software with a community project. Once I sent the following message to the Lua list, which summarizes our approach:

> Lua is open software, but it has never been open developed. That does not mean we do not listen to other people. We read practically every message in the mailing list. Several important features in Lua started or evolved from outside contributions (e.g., metatables, coroutines, and the implementation of closures, to name just some big ones), but ultimately we decide. We do not do this because we consider our judgment better than others'. It is only because we want Lua to be the language we want it to be, not to be the most popular language in the world.
>
> Because of this development style, we prefer not to have a public repository for Lua. We do not want to have to explain every single change we make to the code. We do not want to keep documentation updated all the time. We want freedom to follow strange ideas and then to give up without having to explain every move.

Why do you like to get suggestions and ideas, but not code? One thing that comes to my mind is that maybe writing the code on your own allows you to learn something more about the problem/solution.

Roberto: It is something like that. We like to fully understand what is going on in Lua, so a piece of code is not a big contribution. A piece of code does not explain why it is the way it is, but once we understand the underlying ideas, writing the code is fun we do not want to miss.

Luiz: I think we also had concerns about including third-party code about which we could not guarantee ownership. We certainly did not want to get drowned in the legal processes of having people license their code to us.

Will Lua reach a point where you've added all of the features you want to add, and the only changes are refinements to the implementation (LuaJIT, for example)?

Roberto: I feel we are at such a point now. We have added if not all, most of the features we wanted.

How do you handle smoke testing and regression testing? One of the big benefits I've seen from having an open repository is that you can get people performing automated testing against almost every revision.

Luiz: Lua releases are not that frequent, so when a release does come out, it has been tested a lot. We only release work versions (pre-alpha) when it's already pretty solid, so that people can see what the new features are.

Roberto: We do perform strong regression testing. The point is that, because our code is ANSI C, we usually have very few portability problems. We do not need to test changes in several different machines. I perform all regression tests whenever I change anything in the code, but it is all automated; all I have to do is type `test all`.

When you find a recurrent problem, how do you recognize if the best solution is a local workaround or a global fix?

Luiz: We always try to provide bug fixes as soon as bugs are found. However, since we don't release new versions of Lua frequently, we tend to wait until there are enough fixes to justify the release of a minor version. We leave all improvements that are not bug fixes to major versions. If the issue is complicated (which is quite infrequent), we provide a local workaround in the minor version and a global fix in the next major version.

Roberto: Usually, a local workaround will get you really soon. We should go for a workaround only if it is really impossible to do a global fix—for instance, if a global fix demands a new, incompatible interface.

Would you still design for limited resources now, some years after you started?

Roberto: Sure, our mind is always focused on that. We consider even the order of fields inside C structures to save a few bytes. :)

Luiz: And there are more people putting Lua in small devices today than ever before.

How does the desire for simplicity affect the language design from a user perspective? I think of the support for Lua classes, which reminds me a lot of OO in C in some ways (but much less annoying).

Roberto: Currently we have a rule of "mechanisms instead of policies." That keeps the language simple, but as you said, the user must provide his own policies. This is the case with classes. There are many ways to implement them. Some users love this; others hate it.

Luiz: It does give Lua a do-it-yourself flavor.

Tcl took a similar approach, but it led to fragmentation, as every library or shop had its own approach. Is fragmentation less of an issue because of Lua's intended purpose?

Roberto: Yes. Sometimes it is a problem, but for many kinds of uses (e.g., games) this is not an issue. Lua is mostly used embedded in some other application, and so the application provides a firmer framework for unifying programming styles. You have Lua/Lightroom, Lua/WoW, Lua/Wireshark—each has its own internal culture.

Do you consider Lua's "we provide mechanisms" style of malleability a tremendous benefit then?

Roberto: Not exactly. As most other things, it is a compromise. Sometimes it is very useful to have policies ready for use. "We provide mechanisms" is quite flexible, but needs more work and brings fragmentation of styles. It is also quite economic.

Luiz: On the other hand, sometimes it's hard to explain this to users. I mean, to make them understand what the mechanisms are and what the rationale for them is.

Does that work against code sharing between projects?

Roberto: Yes, frequently. It has hindered the growth of independent libraries, too. For instance, WoW has tons of libraries (they even have an implementation for the traveling salesman problem using genetic programming), but nobody uses that outside WoW.

Do you worry that Lua has splintered somewhat into WoW/Lua, Lightroom/Lua, etc., because of this?

Luiz: We do not worry: the language remains the same. The available functions differ. I guess these applications benefit from this in some ways.

Are serious Lua users writing their own dialects on top of Lua?

Roberto: Maybe. At least we do not have macros. I guess with macros you could create a real new dialect.

Luiz: Not a language dialect per se, but a dialect as domain-specific language implemented with functions, yes. That was the goal of Lua. When Lua is used just for data files, it can look like it's a dialect, but of course they are just Lua tables. There are some projects that do macros, more or less. I recall metalua, for instance. This is a problem with Lisp.

Why did you choose to provide extensible semantics?

Roberto: It started as a way to provide OO features. We did not want to add OO mechanisms to Lua, but users wanted them, so we came up with this idea of providing enough mechanisms for users to implement their own OO mechanisms. We still think this was a good decision. Although this makes OO programming in Lua more difficult for beginners, it brings a lot of flexibility to the language. In particular, when we use Lua mixed with other languages (a Lua hallmark), this flexibility allows the programmer to fit Lua's object model with the object model of the external language.

How does the current environment of hardware, software, services, and networks differ from the environment in which your system was originally designed? How do these changes affect your system and call for further adaptations?

Roberto: Because Lua aims to a very high degree of portability, I would say that the current "environments" are not that different from old ones. For instance, when we started the development of Lua, DOS/Windows 3 were 16-bit machines; some old machines were still 8 bit. Currently we do not have 16-bit desktops, but several platforms where Lua is used (embedded systems) are still 16 bits or even 8 bits.

One big change was in C. When we started Lua, back in 1993, ISO (ANSI) C was not yet as established as it is today. Many platforms still used K&R C, and many applications had some complex scheme of macros to be compiled with K&R C and with ANSI C, the main difference being the declaration of function headers. At that time, it was a bold decision to stick with ANSI C.

Luiz: And we still haven't felt the need to move to C99. Lua is implemented in C89. Perhaps we'll have to use some parts of C99 (especially the new size-specific types) if glitches surface in the transition to 64-bit machines, but I don't expect any.

If you could build the Lua VM all over again, would you stick with ANSI C, or do you wish there were a better language for low-level cross-platform development?

Roberto: No. ANSI C is the most portable language I know (currently).

Luiz: There are excellent ANSI C compilers out there, but even using their extensions does not give us much improved performance.

Roberto: It is not easy to improve ANSI C and keep its portability and performance.

This is C89/90, by the way?

Roberto: Yes. C99 is not very well established yet.

Luiz: Plus I'm not sure C99 would bring us many additional features. I am especially thinking about labeled `goto`s available in gcc as an alternative to `switch` (in the main switch of the vm execution).

Roberto: This is something that could improve performance in many machines.

Luiz: We tested it early on, and someone else tested it recently, and the gains are not spectacular.

Roberto: In part because of our register-based architecture. It favors less opcodes with more work in each one. This decreases the impact of the dispatcher.

Why did you build a register-based VM?

Roberto: To avoid all those getlocal/setlocal instructions. We also wanted to play with the idea. We thought that, if it did not work well, we could at least write some papers about it. In the end, it worked quite well, and we wrote only one paper. :)

Does running in a VM help with debugging?

Roberto: It does not "help"; it changes the whole concept of debugging. Anyone who has ever debugged programs in both compiled and interpreted languages (e.g., C versus Java) knows they are miles apart. A good VM makes the language safe, in the sense that errors can always be understood in terms of the language itself, not in term of the underlying machine (e.g., segmentation fault).

When a language is platform-independent, how does this affect the debugging?

Roberto: Usually it eases debugging, because the more platform-independent a language is, the more it needs a solid abstract description and behavior.

Considering that we are humans and we know for sure we are going to make mistakes when writing software, have you ever thought about which features you needed to add or remove to the language just to aid the debugging phase?

Roberto: Sure. A first step to aid debugging is good error messages.

Luiz: Error messages in Lua have improved since the earlier versions. We have moved from the dreaded "call expression not a function" error message, which existed up until Lua 3.2, to much better error messages such as "attempt to call global 'f' (a nil value)". Since Lua 5.0, we use symbolic execution of the bytecode to try and provide useful error messages.

Roberto: In the design of the language itself, we always try to avoid constructs with complex explanations. If it is hard to understand, it is harder to debug.

What is the link between the design of the language and the design of programs written with that language?

Roberto: At least for me, a main component when designing a language is *user cases*, that is, considerations about how users would use each feature and the combination of features of the language. Of course programmers will always find new ways to use a language, and a good language should allow unanticipated usages, but the "normal" use of the language follows what the designers intended when they created the language.

How much does the implementation of the language affect the design of the language?

Roberto: This is a two-way street. The implementation has a huge impact on the language: we should not design what we cannot implement efficiently. Some people forget it, but efficiency is always a (or the) main constraint in the design of any software. However the design also may have a huge impact on the implementation. At first sight, several distinctive aspects of Lua come from its implementation (small size, good API with C, portability), but the design of Lua plays a key role to enable such implementation.

I read in one of your papers that "Lua uses a handwritten scanner and a handwritten recursive descent parser." How did you start thinking about the idea of building a parser by hand? Was it clear since the beginning that it could have been much better than the yacc-generated one?

Roberto: The first versions of Lua used both lex and yacc, but one of the main original goals of Lua was to be used as a data-description language, not unlike XML.

Luiz: But much earlier.

Roberto: Soon people started using Lua in data files with several megabytes, and the lex-generated scanner quickly became a bottleneck. It is quite easy to write a good scanner by hand and that single change improved Lua performance by something like 30%.

The decision to change from yacc to a handwritten parser came much later, and was not that easy. It started with problems with the skeleton code that most yacc/bison implementations use.

They were not quite portable at the time (e.g., several used `malloc.h`, a non-ANSI C header), and we did not have a good control of their overall quality (e.g., how they handled stack overflows or memory-allocation errors), and they were not reentrant (in the sense of calling the parser during parsing). Also, a bottom-up parser is not as good as a top-down one when you want to generate code on the fly, as Lua does, because it is difficult to handle "inherited attributes." After we made the change, we saw that our handwritten parser was a little faster and smaller than the yacc-generated one, but that was not a main reason for the change.

Luiz: A top-down parser also allows better error messages.

Roberto: However I would never recommend a handwritten parser of any kind for a language without a mature syntax. And for sure LR(1) (or LALR or even SRL) is much more powerful than LL(1). Even for a simple syntax like Lua's, we had to make some tricks to have a decent parser. For instance, the routines for binary expressions do not follow the original grammar at all, but instead we use a smart recursive priority-based approach. In my compiler classes, I always recommend yacc to my students.

Do you have any interesting anecdote from your teaching experience?

Roberto: When I started teaching programming, the main computer facility for our students was a mainframe. Once it happened that a program assignment from a very good group failed even to compile. I talked to them, and they swore they tested their program carefully, with several test cases, and it was working OK. Of course they and I were using exactly the same environment, the mainframe. The mystery remained until a few weeks later, when I learned that the Pascal compiler had been upgraded. The upgrade happened between they finished their task and I started correction. Their program had a very small syntactic error (an extra semicolon, if I recall correctly) that the old compiler did not detect!