Preface

Currently, many programming languages are concerned with how to help you write programs with hundreds of thousands of lines. For that, they offer you packages, namespaces, complex type systems, a myriad of constructions, and thousands of documentation pages to be studied.

Lua does not try to help you write programs with hundreds of thousands of lines. Instead, Lua tries to help you solve your problem with only hundreds of lines, or even less. To achieve this aim, Lua relies on *extensibility*, like many other languages. Unlike most other languages, however, Lua is easily extended not only with software written in Lua itself, but also with software written in other languages, such as C and C++.

Lua was designed, from the beginning, to be integrated with software written in C and other conventional languages. This duality of languages brings many benefits. Lua is a tiny and simple language, partly because it does not try to do what C is already good for, such as sheer performance, low-level operations, or interface with third-party software. Lua relies on C for those tasks. What Lua does offer is what C is not good for: a good distance from the hardware, dynamic structures, no redundancies, ease of testing and debugging. For that, Lua has a safe environment, automatic memory management, and great facility to handle strings and other kinds of data with dynamic size.

More than being an extensible language, Lua is also a *glue language*. Lua supports a component-based approach to software development, where we create an application by gluing together existing high-level components. Usually, these components are written in a compiled, statically typed language, such as C or C++; Lua is the glue that we use to compose and connect those components. Usually, the components (or objects) represent more concrete, low-level concepts (such as widgets and data structures) that are not subject to many changes during program development and that take the bulk of the CPU time of the final program. Lua gives the final shape of the application, which will probably change a lot during the life cycle of the product. However, unlike other glue technologies, Lua is a full-fledged language as well. Therefore, we can use Lua not only to glue components, but also to adapt and reshape them, or even to create whole new components.

Of course, Lua is not the only scripting language around. There are other languages that you can use for more or less the same purposes, such as Perl, Tcl, Ruby, Forth, and Python. The following features set Lua apart from these

languages; although other languages share some of these features with Lua, no other language offers a similar profile:

- *Extensibility:* Lua's extensibility is so remarkable that many people regard Lua not as a language, but as a kit for building domain-specific languages. Lua has been designed from scratch to be extended, both through Lua code and through external C code. As a proof of concept, it implements most of its own basic functionality through external libraries. It is really easy to interface Lua with C/C++ and other languages, such as Fortran, Java, Smalltalk, Ada, and even with other scripting languages.

- *Simplicity:* Lua is a simple and small language. It has few (but powerful) concepts. This simplicity makes Lua easy to learn and contributes for a small implementation. Its complete distribution (source code, manual, plus binaries for some platforms) fits comfortably in a floppy disk.

- *Efficiency:* Lua has a quite efficient implementation. Independent benchmarks show Lua as one of the fastest languages in the realm of scripting (interpreted) languages.

- *Portability:* When we talk about portability, we are not talking about running Lua both on Windows and on Unix platforms. We are talking about running Lua on all platforms we have ever heard about: NextStep, OS/2, PlayStation II (Sony), Mac OS-9 and OS X, BeOS, MS-DOS, IBM mainframes, EPOC, PalmOS, MCF5206eLITE Evaluation Board, RISC OS, plus of course all flavors of Unix and Windows. The source code for each of these platforms is virtually the same. Lua does not use conditional compilation to adapt its code to different machines; instead, it sticks to the standard ANSI (ISO) C. That way, usually you do not need to adapt it to a new environment: If you have an ANSI C compiler, you just have to compile Lua, out of the box.

A great part of the power of Lua comes from its libraries. This is not by chance. One of the main strengths of Lua is its extensibility through new types and functions. Many features contribute to this strength. Dynamic typing allows a great degree of polymorphism. Automatic memory management simplifies interfaces, because there is no need to decide who is responsible for allocating and deallocating memory, or how to handle overflows. Higher-order functions and anonymous functions allow a high degree of parametrization, making functions more versatile.

Lua comes with a small set of standard libraries. When installing Lua in a strongly limited environment, such as embedded processors, it may be wise to choose carefully which libraries you need. Moreover, if the limitations are hard, it is easy to go inside the libraries' source code and choose one by one which functions should be kept. Remember, however, that Lua is rather small

(even with all standard libraries) and in most systems you can use the whole package without any concerns.

Audience

Lua users typically fall into three broad groups: those that use Lua already embedded in an application program, those that use Lua stand alone, and those that use Lua and C together.

Many people use Lua embedded in an application program, such as CGILua (for building dynamic Web pages) or LuaOrb (for accessing CORBA objects). These applications use the Lua–C API to register new functions, to create new types, and to change the behavior of some language operations, configuring Lua for their specific domains. Frequently, the users of such applications do not even know that Lua is an independent language adapted for a particular domain; for instance, CGILua users tend to think of Lua as a language specifically designed for the Web.

Lua is useful also as a stand-alone language, mainly for text-processing and one-shot little programs. For such uses, the main functionality of Lua comes from its standard libraries, which offer pattern matching and other functions for string handling. We may regard the stand-alone language as the embedding of Lua into the domain of string and (text) file manipulation.

Finally, there are those programmers that work on the other side of the bench, writing applications that use Lua as a library. Those people will program more in C than in Lua, although they need a good understanding of Lua to create interfaces that are simple, easy to use, and well integrated with the language.

This book has much to offer to all those people. The first part covers the language itself, showing how we can explore all its potential. We focus on different language constructs and use numerous examples to show how to use them for practical tasks. Some chapters in this part cover basic concepts, such as control structures. But there are also advanced (and original) topics, such as iterators and coroutines.

The second part is entirely devoted to tables, the sole data structure in Lua. Its chapters discuss data structures, persistence, packages, and object-oriented programming. There we will unveil the real power of the language.

The third part presents the standard libraries. This part is particularly useful for those that use Lua as a stand-alone language, although many other applications also incorporate all or part of the standard libraries. This part devotes one chapter to each standard library: the mathematical library, the table library, the string library, the I/O library, the operating system library, and the debug library.

Finally, the last part of the book covers the API between Lua and C, for those that use C to get the full power of Lua. This part necessarily has a flavor quite different from the rest of the book. There we will be programming in C, not in Lua; therefore, we will be wearing a different hat. For some readers, the discussion of the C API may be of marginal interest; for others, it may be the

most relevant part of this book.

Other Resources

The reference manual is a must for anyone that wants to really learn any language. This book does not replace the Lua reference manual. Quite the opposite, they both complement each other. The manual only describes Lua. It shows neither examples nor a rationale for the constructs of the language. On the other hand, it describes the whole language; this book skips some seldom-used dark corners of the language. Moreover, the manual is the authoritative document about Lua. Wherever this book disagrees with the manual, trust the manual. To get the manual and more information about Lua, visit the Lua site at `htpp://www.lua.org`.

You can also find useful information at the Lua users site, kept by the community of users at `htpp://lua-users.org`. Among other resources, it offers a tutorial, a list of third-part packages and documentation, and an archive of the official Lua mailing list. It may be useful to check also the book's web page:

```
http://www.inf.puc-rio.br/~roberto/book/
```

There you can find an updated errata, code for some of the examples presented in the book, and some extra material.

This book describes Lua 5.0. If you are using a more recent version, check the corresponding manual for eventual differences between versions. If you are using an older version, this is good time to upgrade.

A Few Typographical Conventions

The book encloses "`literal strings`" between double quotes and single characters, like '`a`', between single quotes. Strings that are used as patterns are also enclosed between single quotes, like '`[%w_]*`'. The book uses a courier font both for little `chunks of code` and for `identifiers`. Larger chunks of code are shown in display style:

```
-- program "Hello World"
print("Hello World")         --> Hello World
```

The notation `-->` shows the output of a statement or, eventually, the result of an expression:

```
print(10)     --> 10
13 + 3        --> 16
```

Because a double hyphen (`--`) starts a comment in Lua, there is no problem if you include those annotations in your programs. Finally, the book uses the notation `<-->` to indicate that something is equivalent to something else:

```
this     <-->     that
```

That is, there is no difference to Lua whether you write `this` or `that`.

About the Book

I started writing this book in the winter of 1998. (Here, in the southern hemisphere, that means the middle of the year. And "winter" is more like a mild autumn.) At that time, Lua was still in version 3.1. Since then, Lua went through two big changes, first to version 4.0, in 2000, then to version 5.0, in 2003.

It is quite obvious that those changes had a big impact on the book. Some parts lost their raison d'être, such as the detailed explanation around the complexity of upvalues. Whole chapters were rewritten, such as those about the C API, and whole chapters were created, such as the one about coroutines.

What is not obvious, however, is the big impact that the writing of this book had on the evolution of Lua. Not by chance, some of the biggest changes in the language were in areas not yet covered by the book at the time of the change. As I worked through the book, sometimes I suddenly got stuck in a chapter. I could not figure out how to start or even how to motivate it. It is when you try to explain how to use something that you better feel how easy it is to use it (or not). So, those difficulties were strong hints that some things in Lua needed improvement. Other times I succeeded in writing a chapter, only to discover, later, that nobody could understand or agree with what I wrote. Frequently it was my fault (as I writer), but occasionally we spotted another corner of the language that deserved some improvement. (For instance, the transition from upvalues to lexical scoping was triggered by complaints over a feeble attempt, in an earlier draft of this book, to describe upvalues as a kind of lexical scoping.)

The changes of the language deferred the completion of this book; now the completion of this book will probably defer significant changes in the language. There are at least two reasons for that: First, Lua 5.0 is cleaner and more mature than earlier versions of the language (partially thanks to the book). Second, the book adds weight to the culture around the language and therefore increases its inertia. This cultural-weight increase is the first of my main goals with this book. My second main goal is to increase even more the spread of Lua.

Acknowledgments

Several people helped me to write this book. Luiz Henrique de Figueiredo and Waldemar Celes, co-authors of Lua, helped to improve Lua, therefore making my job here easier. Luiz Henrique also helped with the interior book design. Noemi Rodriguez, André Carregal, Diego Nehab, and Gavin Wraith reviewed drafts of this book and provided invaluable suggestions. Renato Cerqueira, Carlos Cassino, Tomás Guisasola, Joe Myers, and Ed Ferguson also provided important suggestions. Alexandre Nakonechnyj designed the book cover and also helped with the interior book design. Rosane Teles prepared the Cataloging-in-Publication (CIP) data. My thanks to you all.