The Programming Language Lua

Jenny Carlsson, Sakari Nousiainen

Background
Lua is an open scripting language with its origins in Brazil. The language was founded in University of Rio de Janeiro 1993 and is currently licensed under very permissive MIT license. As it is written in ANSI C, it is highly portable and available at a wide variety of platforms. The main principles of Lua is simplicity and portability for different platforms, and it is intended to be of a small size. The principal applications are gaming, embedded devices and scripting applications. Lua is often used as scripting language, working concurrently with other programming languages. This means that either Lua calls functions from another language for example C++ or the other way around.

Lua is relatively widely used in software industry and has been used in successful games like World of Warcraft, Crysis and Baldur's Gate. In addition it is actively used by Adobe products and VLC open source video player. Therefore it can be said that Lua is not a niche language of geeks, but having a significant userbase and so it has proved to valuable.

Core features
Lua can be found as a small library, pre-compiled for your platform, or its source can be downloaded and be compiled at any device with a C compiler. With its 79 functions, the API is simple, and - according to its maintainers - has a gentle learning curve. However it has been criticized [1] that even if the language is quite simple is does not imply that non-coders, artists could work with the language easily.

The language is said to be a multi-paradigm language, because while mainly being a procedural language - having data and behaviour separated - it also borrows paradigms from functional and object-based programming (OOP). Class-like inheritance pattern familiar from C++ OOP can be acquired using metatables in Lua, which tell the table-object to look for missing methods from higher level tables. As such, the table type is very flexible and can be used to implement for example namespaces.

Syntax
The whole syntax of language can be presented [2] as an extended Back-Naur From (BNF) with just approximately one page of definitions. The syntax of Lua is quite non-restrictive, as line breaks do not affect interpretation. For clarity, extra semicolons can optionally be added after any statement. For variable naming, most things are allowed, within the constraints of the chosen locale, i.e. français could be a variable name in a latin alphabet. Also, a variable name can not start with a number, neither with an underscore, as these are reserved for internal usage in the Lua library. Lua is case-sensitive.
Types
As many other interpreted languages, Lua has dynamic typing of variables. The eight basic types that are available in Lua are nil, boolean, number, string, userdata, function, thread, and table. A variable of type nil is, as the name implies, non-existent. Actually, it is possible to delete a global variable by assigning the value nil to it. Otherwise Lua's automatic garbage collector handles deletion of unused variables. The number type can handle real numbers, i.e. double-precision floating-point numbers. A practical way to input a number in Lua is to use scientific notation, i.e. exponent form: 3.9e10. Clearly the boolean type can have values true and false, but in an expression, only false and nil are false. For programmers originating from different languages, it is important to notice that this means that zero and the empty string are actually interpreted as true. Another thing to notice, is that the string type is immutable in Lua. That is, when you change a character in a string, a new string is created.

In Lua, there is only one data structure mechanism; the table. The table in Lua is an associative array, i.e. an array that can be indexed with any value of the language, with the exception of nil. The table is a very flexible structure that can be used to implement more or less every basic data structure usually used in programming. If an index is not provided when a new element is added, the next available is used, beginning from 1. The function ipairs can be used to iterate through the elements of the array. Lua implements a structure named metatables that could be said to be parent object of every variable (including functions etc.). Metatables can contain events and methods which describe the behaviour of child-object in lower level. All basic types in Lua share same metatable which defines how specific type behaves. Each table and userdata however have their own metatables.

Functions
The functions in Lua are first-class values, meaning that functions can be stored in variables and can be passed to and returned from functions just as any other value. The functions also have proper lexical scoping: Just like in functional programming, a nested function can use variables from enclosing functions, eliminating the need to pass all values needed. Care needs to be taken though, if local functions are used. In Lua, all functions are anonymous, so the syntax function f (params) body end really is the same as f = function (params) body end.

Functions in Lua can be used like procedures (as a function in Lua does not need to include the return-keyword) or functions that return values. Also, a function can return any number of values, and these values can be taken advantage of by storing them using multiple assignment. If a function f returns length, width and height, a function call may look like: l,w,h=f(). Any extra variables gets the value nil, and if there are too few, some or all of the results are discarded. This principles apply to all similar situations, such as in function calls: A function can be called with any number of arguments, for if there are too many arguments, the last ones are discarded and any parameters not used gets the value nil. One useful feature in Lua is the variable number parameter list. If the function definition has three dots in its parameter list, such as (...) or (param, ...), the caller can specify any number of arguments. The function can access the parameters in a hidden parameter of type table.
Another good feature of Lua, saving space in certain situations, is that it does proper tail calls. When a function calls another as the last thing it does before returning, the function is taken out of the memory stack, as it is not needed any more. Therefore, there is no limit to the amount of nested function calls that can be made in this manner.

Though Lua is a very small language, it can be expanded to great lengths as functions in Lua can also be defined in C. In fact, functions can be defined in any language supported by the host application.

**Control structures**

Lua has four different control structures, all familiar from most other programming languages: **if-then-else**, **while**, **repeat** and **for**. All these must be followed by the keyword **end**. For nested if statements, **elseif** can be used in order to eliminate the need for many **ends**. Of the **for** loop, there are two variants: the numeric and the generic for-loop. The numeric is like in c/c++, where the loop executes from the value of one expression to another with the step length to increment the loop variable. The generic for loop is traversed with the help of an iterator function. The **for** loop in Lua is not guaranteed to function if the loop variable is changed manually inside the loop. The control structures can be escaped by **return** or **break**. These can only be at the last statement of a block. If, for example, a return is needed somewhere else in a block, a surplus block can be created by bracketing the statement with the keyword-pair **do-end**. This can also be used to further limit the scope of a local variable.

**Other notes and GC**

Lua has some neat tricks not available in many languages. The multiple assignment mentioned earlier can be used to swap two values, i.e. \( x, y = y, x \). However, Lua also has some trickier things that needs to be noted. As Lua uses a reference for the types **table**, **userdata** and **function**, the user must note that when comparing these, Lua compares the references. In other words, the two values are equal only when the objects are the same object.

As in most script languages garbage collection (GC) is implemented in lua. It has two parameters **pause and step multiplier**, which control how often and at how large chunks the memory is freed. User is capable of changing these parameters from the code. It can be questionable should such GC controlling be necessary, but at least it allows some flexibility. Lua also let's user to implement GC metamethods, which can be used to synchronize memory management with external languages, such as C++. A weak table structure can be used as non-owning references to objects, meaning that they don't have an effect on object lifetime.

Though this text has so far mentioned mostly positive aspects of the language, there are some problems that should be considered. Lua is a low level language, and as such, it needs quite many calls to achieve a task. This is also noticed in the fact that there is no direct mapping of the more complex types data structures. Also some users can find the stack oriented programming style of Lua to be confusing.
References

For the interested reader, more information can naturally be found at Lua’s website, where the first version of the book “Programming with Lua” by Roberto Ierusalimschy also can be found. This book was the main reference material used for this essay. Some material has also been borrowed from a series of slides by R. Ierusalimschy from the University PUC of Rio de Janeiro, titled “Learn Lua in X minutes”.


http://www.lua.org/pil/contents.html