The Design of Lua

Roberto Ierusalimschy
The design of a language involves many trade-offs, and we need explicit goals and priorities to settle these trade-offs. Different languages choose different goals, and therefore settle these trade-offs in different directions. Like any tool, no language is good for everything.
Some PL Trade-offs

• Safety versus flexibility
  – what you cannot do!
  – type checking
  – memory management

• Readability versus conciseness
  – Perl: write once, read nowhere

• Performance versus abstractions

• Libraries versus portability
Some PL Trade-offs

- Flexibility versus good error messages
  - Haskell
- Simplicity versus expressiveness
We need explicit goals to solve trade-offs!
Lua Goals

- Portability
- Simplicity
- Small size
- Scripting
Portability

• Runs on most platforms we ever heard of
  – Posix (Linux, BSD, etc.), OS X, Windows, Android, iOS, Arduino, Raspberry Pi, Symbian, Nintendo DS, PSP, PS3, IBM z/OS, etc.
  – written in ANSI C

• Runs inside OS kernels
  – NetBSD, Linux

• Written in ANSI C, as a free-standing application
Simplicity

Reference manual with less than 100 pages (proxy for complexity)

Documents the language, the libraries, and the C API.
Size

![Graph showing the growth of Lua versions from Lua 1.0 to Lua 5.3 over time. The x-axis represents the years from 1990 to 2020, and the y-axis represents the size in LOC. Each Lua version is marked with a corresponding point on the graph.]

- Lua 1.0
- Lua 4.0
- Lua 5.0
- Lua 5.1
- Lua 5.2
- Lua 5.3

LOC and LLOC are represented by different markers and colors.
Scripting

• Scripting language x dynamic language
  – scripting emphasizes inter-language communication

• Program written in two languages
  – a scripting language and a system language

• System language implements the hard parts of the application
  – algorithms, data structures
  – little change

• Scripting *glues* together the hard parts
  – flexible, easy to change
Lua and Scripting

- Lua is implemented as a library
- Lua has been designed for scripting
- Good for *embedding* and *extending*
- Embedded in C/C++, Java, Fortran, C#, Perl, Ruby, Python, etc.
Scripting in Grim Fandango

“[The engine] doesn't know anything about adventure games, or talking, or puzzles, or anything else that makes Grim Fandango the game it is. It just knows how to render a set from data that it's loaded and draw characters in that set. […]

“The real heroes in the development of Grim Fandango were the scripters. They wrote everything from how to respond to the controls to dialogs to camera scripts to door scripts to the in-game menus and options screens. […]

“A TREMENDOUS amount of this game is written in Lua. The engine, including the Lua interpreter, is really just a small part of the finished product.”

Bret Mogilefsky
Goals: Impact on Uses
Embedded Systems

Samsung (TVs), Cisco (routers), Logitech (keyboards), Volvo (car panels), Olivetti (printers), Océ (printers), Ginga (middleware for digital TV), Verison (set-top boxes), Texas Instruments (calculators), Huawei (mobiles), Sierra Wireless (M2M devices), NodeMCU (IoT), …
Goals: Impact on Design
“Closures”

• Anonymous functions as first-class values with lexical scoping

• Now more common in non-functional languages, but not that common
  – closing on variables x closing on values
  – other idiosyncrasies

• Few non-functional languages use closures as pervasively as Lua
“Closures”

• Pros
  – simple and well-established concept (lambda calculus!?)
  – powerful and empowering feature
  – easy to interface with other languages

• Cons
  – complex implementation
  – syntax too cumbersome for small functions
Tables

- Associative arrays
  - any value as key: strings, numbers, objects, etc.
- Only data structure mechanism in Lua
- Tables implement many data types in simple and efficient ways
  - sets, arrays, sparse matrices, lists, structures
- Tables in Lua are also used for several other purposes
  - global variables, modules, objects and classes
Tables

• Pros
  – simple semantics
  – powerful
  – easy to interface with other languages

• Cons
  – emulation of other structures are not as good as “the real thing”
  – complex implementation
Exception Handling

• All done through two functions, pcall and error

```lua
try {
    <block/throw>
} catch (err) {
    <exception code>
}
```

```lua
local ok, err = pcall(function ()
    <block/error>
end)
if not ok then
    <exception code>
end
```
Exception Handling

• Pros
  – simple semantics
  – no extra syntax
  – simple to interface with other languages

• Cons
  – verbose
  – try is not cost-free
Iterators

- Old style:

```lua
local inv = {}
table.foreach(t, function (k, v)
    inf[v] = k
end)
```

- New style:

```lua
for w in allwords(file) do
    print(w)
end
```
function allwords (file)
  local line = io.read(file)
  local pos = 1
  return function ()
    while line do
      local w, e = string.match(line, "(%w+)(())", pos)
      if w then
        pos = e
        return w
      else
        line = io.read(file)
        pos = 1
      end
    end
    return nil
  end
end
Iterators

• Pros
  – easy to interface with other languages
  – simple

• Cons
  – cannot traverse nil
  – not so simple as explained
Modules

• Tables populated with functions

```lua
local math = require "math"
print(math.sqrt(20))
```

• Several facilities come for free
  • submodules
  • local names

```lua
local m = require "math"
print(m.sqrt(20))
local f = m.sqrt
print(f(10))
```
Modules

• Pros
  – needs very few new features
  – easy to interface with other languages
  – flexible

• Cons
  – not as good as “the real thing” (regarding syntax)
  – too dynamic (?)
Objects

• first-class functions + tables ≈ objects
• syntactical sugar for methods
  • handles self

```plaintext
function a:foo (x)
  ...
end

a:foo = function (self, x)
  ...
end
```

a:foo(x) ➔ a.foo(a, x)
Delegation

• field-access delegation (instead of method-call delegation)

• when a delegates to b, any field absent in a is got from b
  • a[k] becomes (a[k] or b[k])

• allows prototype-based and class-based objects

• allows single inheritance
Delegation at work

\[
\begin{align*}
\text{k} & = 0 \\
\text{delegate:} & \\
\text{a:foo(x)} & \\
\text{a.foo(a,x)} & \\
\end{align*}
\]
Objects

- **Pros**
  - flexible
  - easy to interface with other languages
  - clear semantics
  - needs few new features

- **Cons**
  - may need some work to get started
  - no standard model (DIY)
Perspective (in the small)

- Tables (associative arrays) and closures are two basic concepts that proved to be extremely flexible and general.
Perspective (in the large)

- No language is truly *general-purpose*
- Any design involves trade-offs
- Different languages prioritize different goals to solve trade-offs
- Lua has a unique set of goals
  - simplicity, portability, scripting
Enter door to LUA Bar