The Design of Lua

Roberto Ierusalimschy

Pontifícia Universidade Católica
do Rio de Janeiro

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*

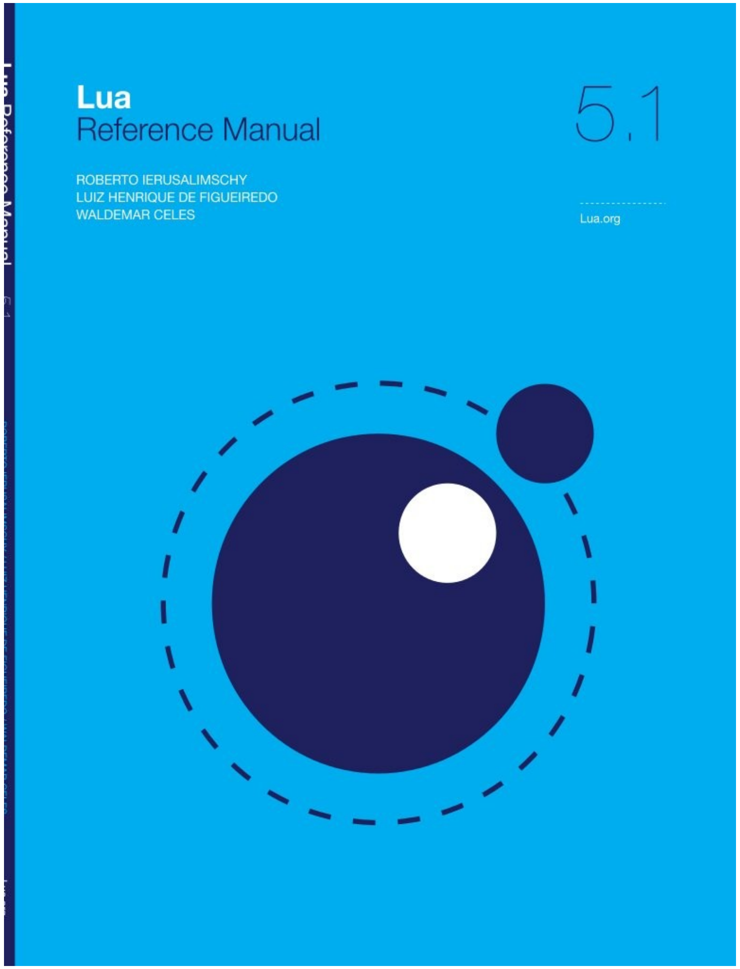
Simplicity

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

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

(spine)

Lua Reference Manual 5.1 ROBERTO IERUSALIMSCHY / LUIZ HENRIQUE DE FIGUEIREDO / WALDEMAR CELES Lua.org



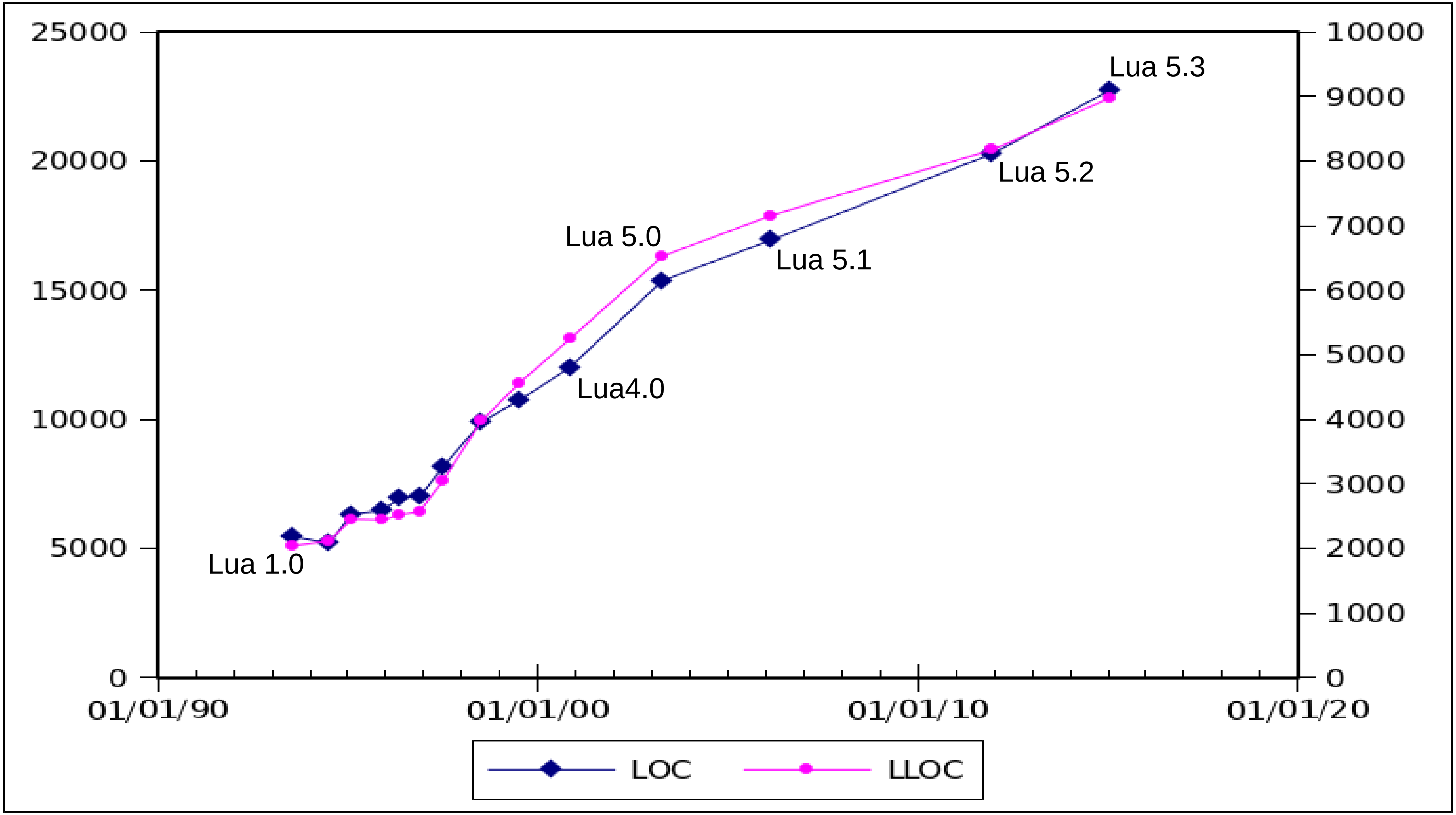
Lua
Reference Manual

5.1

ROBERTO IERUSALIMSCHY
LUIZ HENRIQUE DE FIGUEIREDO
WALDEMAR CELES

Lua.org

Size


25000
20000
15000
10000
5000
0
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
01/01/90
01/01/00
01/01/10
01/01/20
Lua 1.0
Lua4.0
Lua 5.0
Lua 5.1
Lua 5.2
Lua 5.3
LOC
LLOC

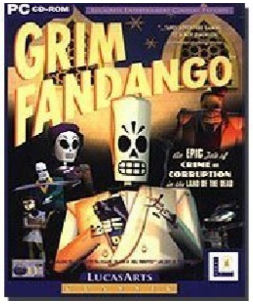
Scripting

- 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. […]

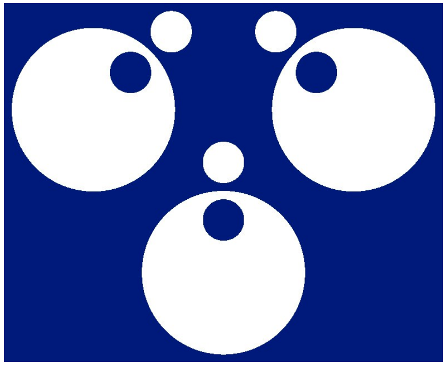
“A TREMENDOUS amount of this game is written in Lua. The engine, including the Lua interpreter, is really just a small part of the finished product.”

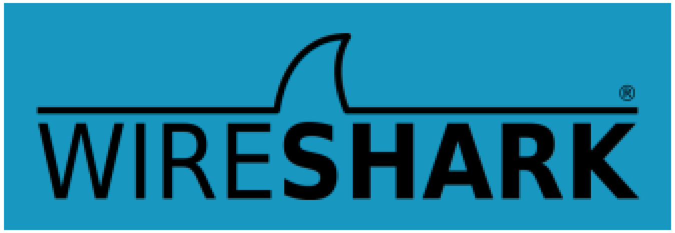
Bret Mogilefsky

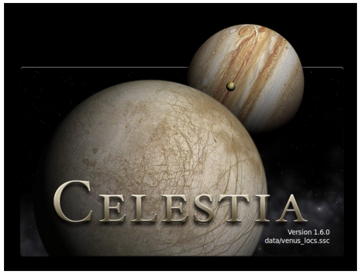
Goals: Impact on Uses

Embedded Systems

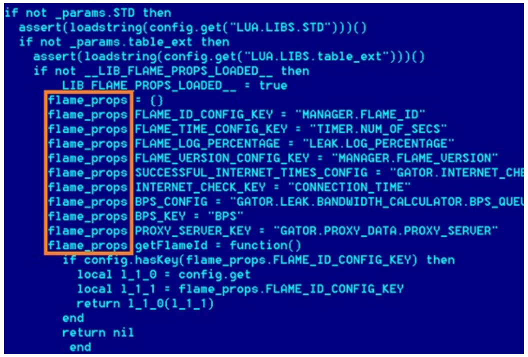
Samsung (TVs), Cisco (routers), Logitech (keyboards), Volvo (car panels), Olivetti (printers), Océ (printers), Ginga (middleware for digital TV), Verison (set-top boxes), Texas Instruments (calculators), Huawei (mobiles), Sierra Wireless (M2M devices), NodeMCU (IoT), …







Fusion 8
The world's most advanced visual effects and motion graphics software, now available for Mac and Windows!
Public Beta Now Available
Download Today!



```
if not _params.STD then
  assert(loadstring(config.get("LUA.LIBS.STD")))()
  if not _params.table_ext then
    assert(loadstring(config.get("LUA.LIBS.table_ext")))()
    if not __LIB_FLAME_PROPS_LOADED__ then
        LIB FLAME PROPS_LOADED__ = true
      flame_props = {}
      flame_props FLAME_ID_CONFIG_KEY = "MANAGER.FLAME_ID"
      flame_props FLAME_TIME_CONFIG_KEY = "TIMER.NUM_OF_SECS"
      flame_props FLAME_LOG_PERCENTAGE = "LEAK.LOG_PERCENTAGE"
      flame_props FLAME_VERSION_CONFIG_KEY = "MANAGER.FLAME_VERSION"
      flame_props SUCCESSFUL_INTERNET_TIMES_CONFIG = "GATOR.INTERNET_CHE
      flame_props INTERNET_CHECK_KEY = "CONNECTION_TIME"
      flame_props BPS_CONFIG = "GATOR.LEAK.BANDWIDTH_CALCULATOR.BPS_QUEU
      flame_props BPS_KEY = "BPS"
      flame_props PROXY_SERVER_KEY = "GATOR.PROXY_DATA.PROXY_SERVER"
      flame_props getFlameId = function()
        if config.hasKey(flame_props.FLAME_ID_CONFIG_KEY) then
          local l_1_0 = config.get
          local l_1_1 = flame_props.FLAME_ID_CONFIG_KEY
          return l_1_0(l_1_1)
        end
        return nil
         end
```

Company of Heroes
Universe at War
Dungeons
Tap Tap Revenge
Blitzkrieg
Monopoly Tycoon
Driver San Francisco
ROBLOX
Requiem Memento Mori
Grim Fandango
Amped
Hulk
Destroy All Humans!
Baldur's Gate
Rail Simulator
World of Warcraft
The Lord of the Rings Online
Far Cry
Impossible Creatures
S.T.A.L.K.E.R.
Blossom
NFL Fever
Soul Ride
Diner Dash 2
Natural Selection 2
Allods Online
There
Brütal Legend
SimCity 4
Angry Birds
SpellForce
Napoleon Total War
Supreme Commander
Battlezone 98 Redux
Top Spin 2
Buzz! Quiz TV
Sonic the Hedgehog
The Witcher
Crackdown
Crimson Steam Pirates
MDK2
Crysis
Supreme Commander Forged Alliance
Homeworld2
Psychonauts
Bubble Ball
Star Wars Empire at War
PlayStation Home
The Incredibles
Links
Dungeons
Brave
The Guild 2
Escape from Monkey Island
Empire Total War
Civilization V
plus+
Crackdown
Heroes V
Dawn of War Dark Crusade
You Are Empty
Fable II

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`

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

```
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:

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

- New style:

```
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

```
local math = require "math"
print(math.sqrt(10))
```

- Several facilities come for free
  - submodules
  - local names

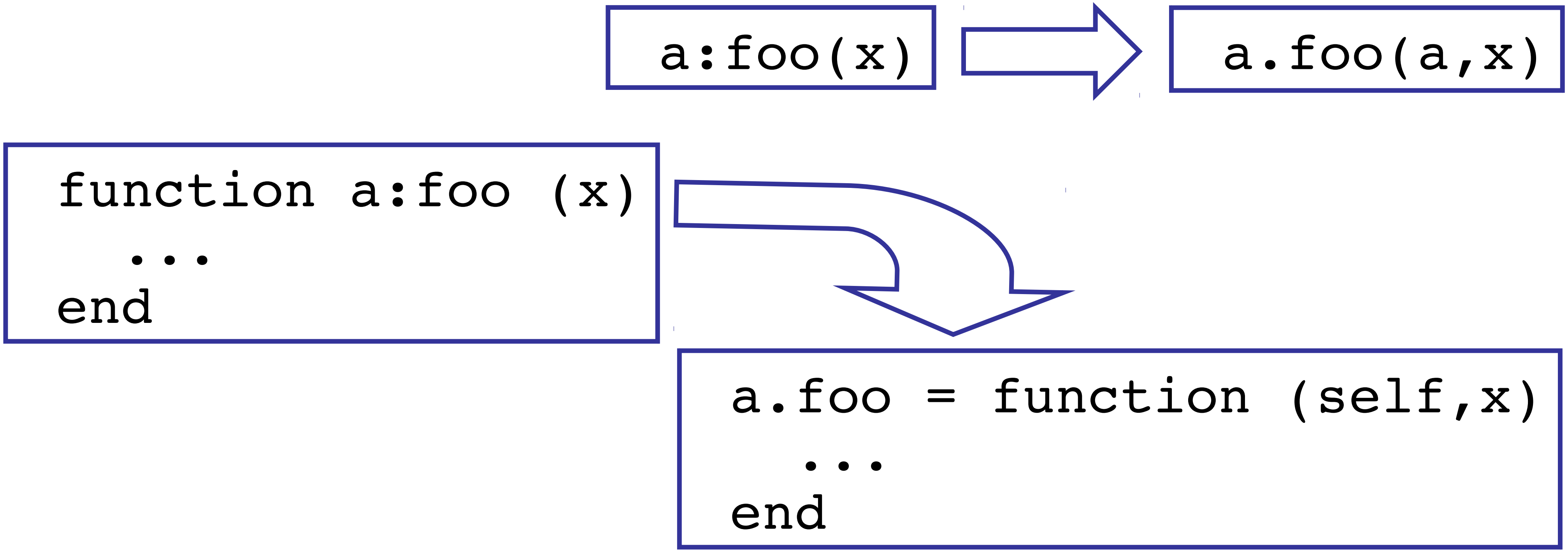
```
local m = require "math"
print(m.sqrt(20))
local f = m.sqrt
print(f(10))
```

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*



```
a:foo(x)
```

```
a.foo(a,x)
```

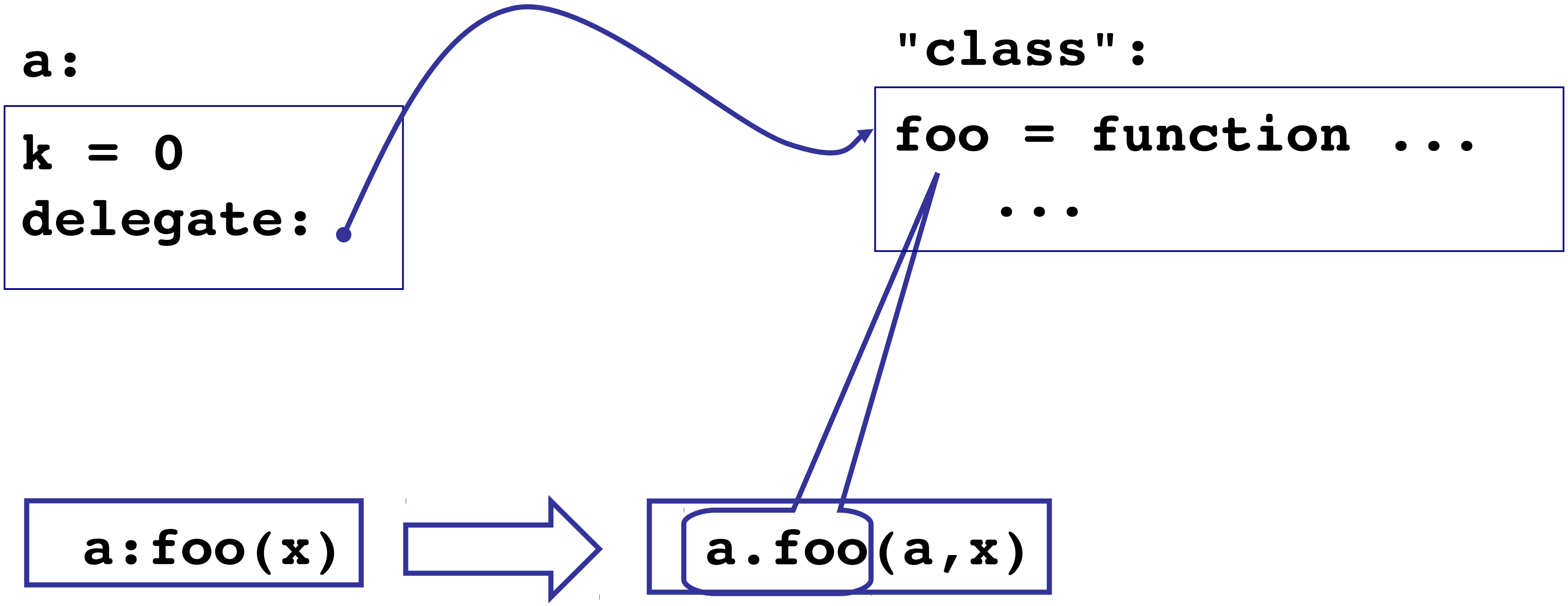
```
function a:foo (x)
  ...
end
```

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

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



```
a:
k = 0
delegate:
```

```
"class":
foo = function ...
   ...
```

```
a:foo(x)
```

```
a.foo(a,x)
```

Objects

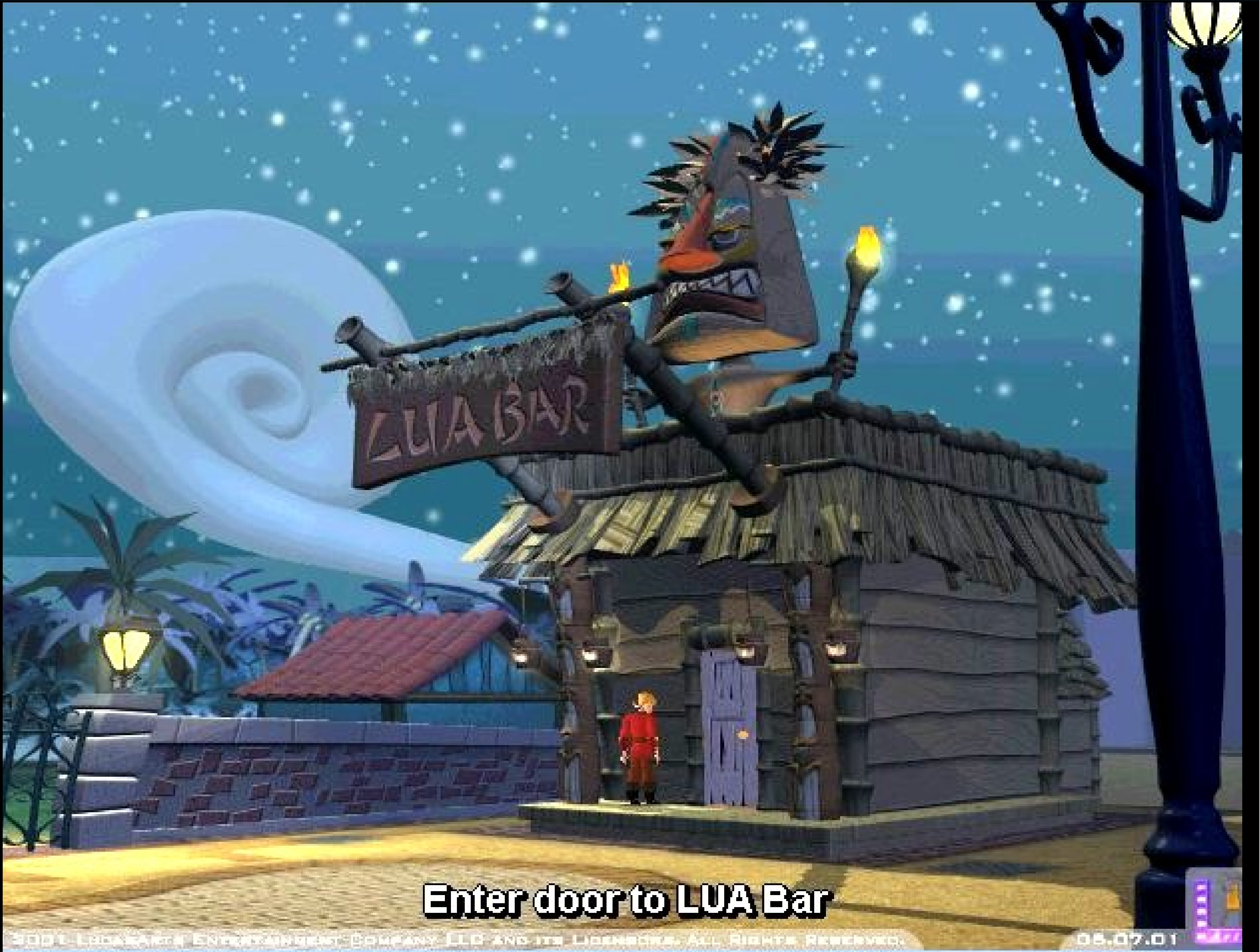
- Pros
  - flexible
  - easy to interface with other languages
  - clear semantics
  - needs few new features
- Cons
  - may need some work to get started
  - no standard model (DIY)

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