
# Introducción a Lua

jvisca@fing.edu.uy / MINA / Robótica Embebida / 2015

# Diseño de Lua

- Objetivos
- Lenguaje
- Implementación

# Diseño de Lua

*Lua is a powerful, fast, lightweight, embeddable scripting language.*

*Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.*

lua.org/about.html

# Objetivos: embebible

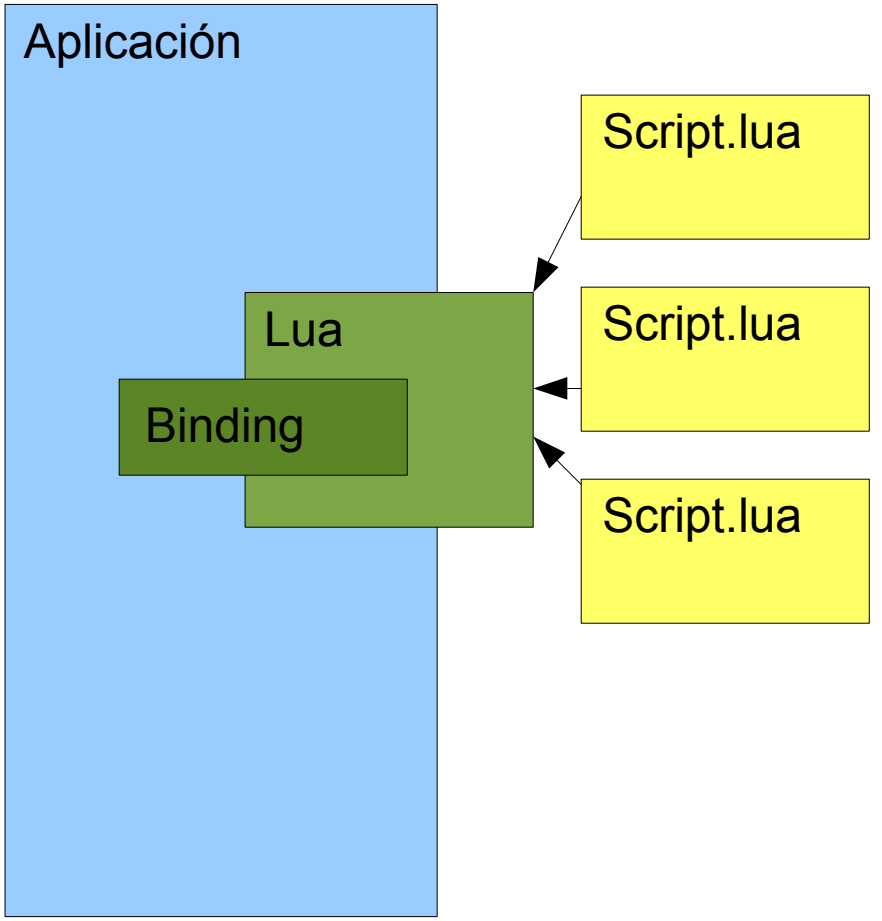
Lenguaje para extender sistemas.

Lenguaje para especificar lógica de aplicación de alto nivel.

Lógica de aplicación ajustada por el usuario.

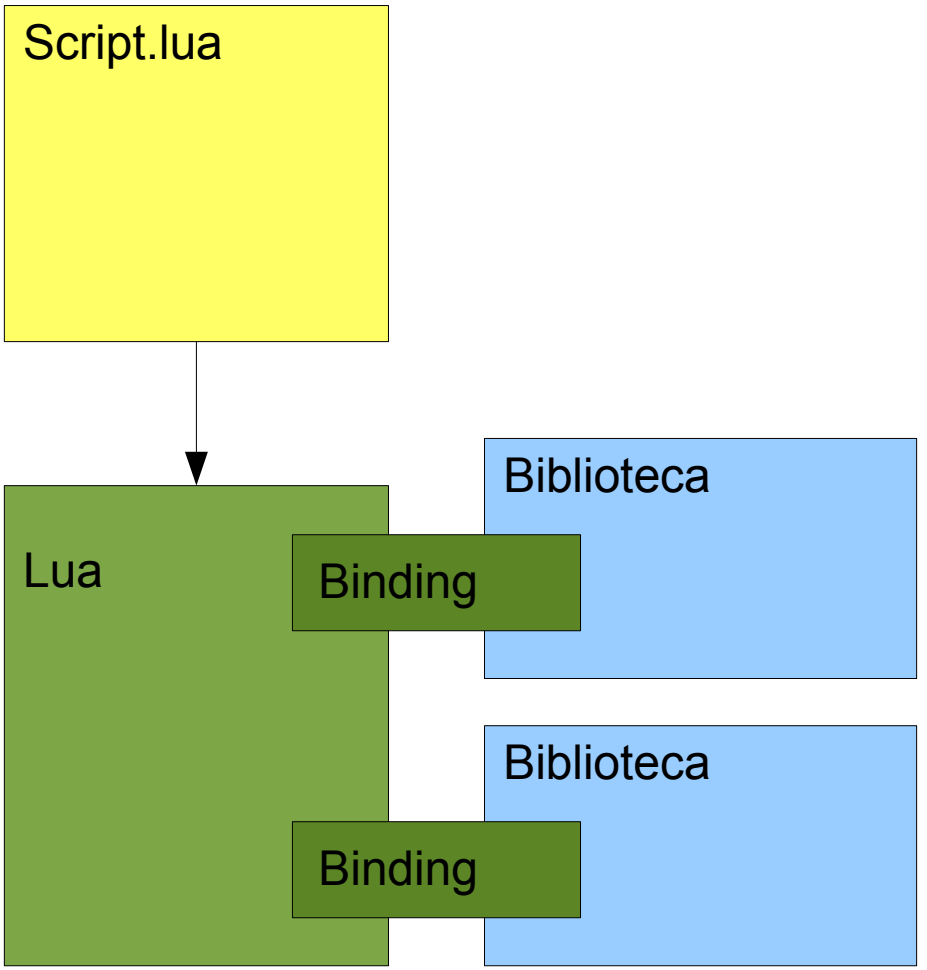
# Embebible, 1er caso

- Extender una aplicación agregando scripting.
  - Archivos de configuración...
- Ejemplos: nmap,World of Warcraft, Wireshark, Adobe Lightroom...



# Embebible, 2do caso

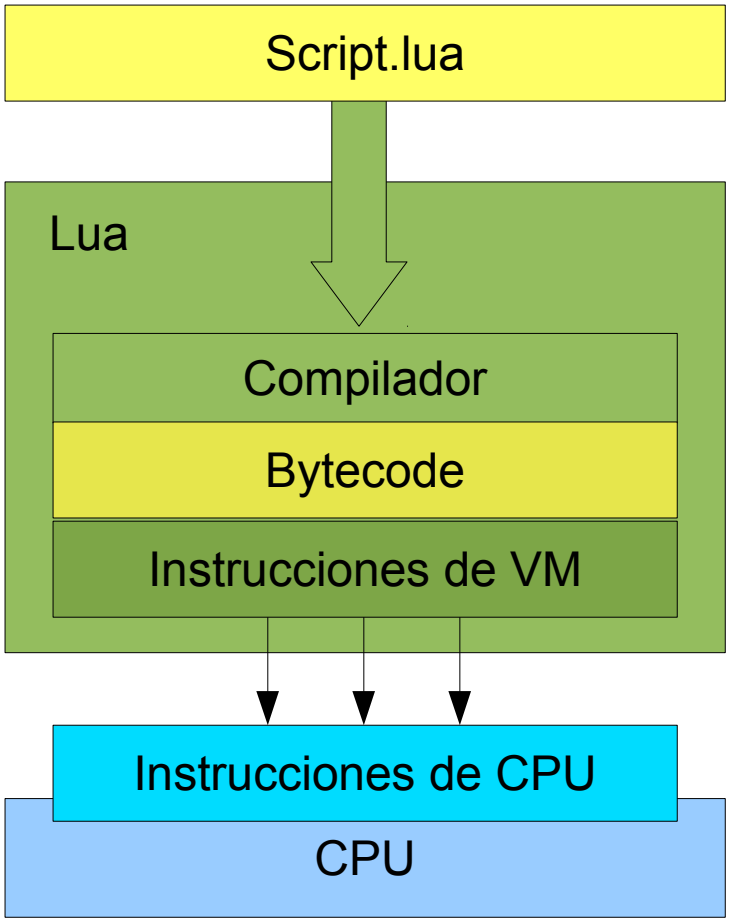
- Interconectar bibliotecas y aplicaciones nativas
- Ejemplos: Toribio!
  - Servicios accesibles por http y protocolo propio
  - Accede a módulos nativos: libusb, sockets
  - Sistema de plugins



# Embebible

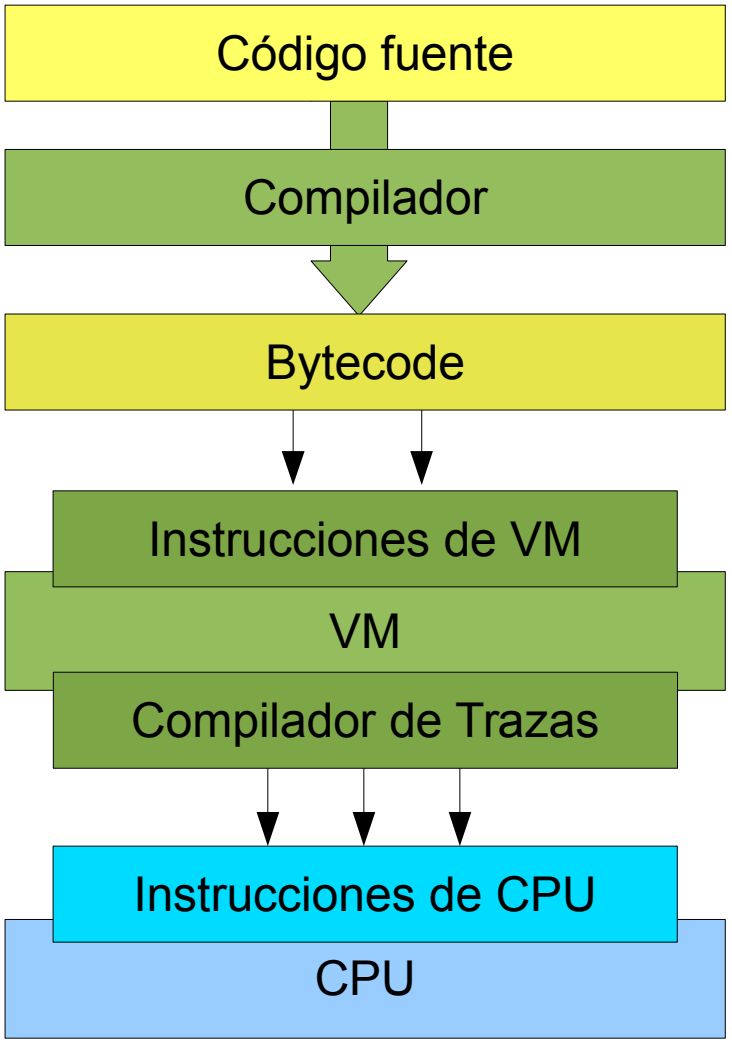
- Permite implementaciones muy eficientes y compactas
  - Lua VM
    - Lua clásico: compacto y portable
  - LuaJIT
    - Rápido! (para x86, ARM y PPC...)
  - LLVM, Metalua,etc.

# LuaVM




- Compilador a bytecode muy rápido
  - En una sola pasada
  - Los programas suelen distribuirse en forma de fuentes, no bytecode (el compilador se incluye junto con la VM)
  - La forma de usar parece un intérprete puro
  - bytecode no portable (!)
- VM simple y eficiente
  - Basado en registros
  - Altamente portable (escrito en ANSI C) y sin dependencias externas
  - Muy pequeño (100-250kb)
- Versión 5.1 & 5.2, 5.3 recién salida.

# LuaJIT




- Ventajas
  - Las de una VM común
  - Velocidad! (puede superar un programa en C)
  - FFI para bindings!
- Dificultades
  - Portabilidad
  - Complejidad
  - Overhead de la compilación JIT

# Lua

```
--calcula el factorial
function fact (n)
   if n == 0 then
      return 1
   else
      return n * fact(n - 1)
   end
end

print( 'Digite un numero:' )
local a=io.read( "*number" ) --lee un numero
print( 'El factorial es:', fact(a) )
```

# Lua: bucles

```
for i=1,10 do
   print("natural",i)
end


for i=0,10,2 do
   print("par",i)
end


for k,v in iter_f do
   print (k,v)
end
```

```
local i=1
while i<=10 do
   print("potencia", i)
   i=i*2
end


repeat
   local x=math.random()
until x>0.9


break
```

# Lua: Tipado Dinámico

- string
- number
- boolean
- table
- nil
- function
- coroutine
- userdata

# Lua: Garbage Collector

- Incremental
- Reemplazable: arquitecturas exóticas suelen proveer su propia gestión de memoria
- Soporta referencias cíclicas
- Fácil de controlar: tablas *weak*
- En general, funciona bien y hay que dejarlo en paz.

# Lua: string

- Strings *internos* e *inmutables*: una sola instancia por cadena de texto
  - Tiempo lineal de creación, tiempo constante de comparación
- ```
  a = "cadena"
  b = 'otra cadena'
  c = [[cadena larga
  y multilinea]]
  d = a .. " y " .. c
  largo = #e
  ```
  ```

# Lua: number

- Un solo tipo!
  - Usualmente *double* (o *float*).
- Pero no importa. Sorprendentemente funciona:
  - Lógica de aplicaciones manipula “números”. El resto es dependiente de la arquitectura y se considera de bajo nivel.
- Si realmente necesita mas tipos, puede usar LNUM, BitOp, LuaJIT
- Lua 5.3 optimiza uso de enteros de forma transparente.

# Lua: casteo automático

- ```
  num = 1
  print("texto" .. num .. "!")
  --> texto1!
  ```
- Pero recuerde: `1` no es igual a `"1"`
- Operador `+` no sobrecargado para strings
- En caso de duda, usar
  - `tonumber()`
  - `tostring()`
  - `type()`
  ```

# Lua: Scoping

- Variables globales o locales
- `a = 10 -- global`
  `local b = 10 -- local`
- Variables globales van al entorno
- Variables locales son un registro de la VM
  - En caso de duda, usar local.
  - Alcance léxico: dentro de la función, estructura de control o bloque `do – end`
- ¿Por qué no local por defecto?
  - Pista: las funciones son anidables
    - http://lua-users.org/wiki/LocalByDefault

```
local a = 1
do
   local a = 2
   print(a)
end
print(a)

→
2
1
```

# Lua: asignación múltiple

- ```
  local a, b, c = 1, 0, "opi"
  a, b = b, a  --intercambiar valores
  ```

- ```
  local function sum_mult (a,b)
    return a+b, a*b
  end
  local sum, mult = sum_mult(2,3)
  ```

- ```
  local function div (a, b)
    if b~=0 then
       return a/b
    else
       return nil, "divide by zero"
    end
  end
  local q=assert( div(2,0) )
  ```
  ```

# Lua: tablas (1)

- Única estructura de datos (!), se usa para implementar todo
- ```
  local t = {}
  t[1] = "primero"
  t["primero"] = true
  ```
- Valor de `nil` representa ausencia
  ```
  t["primero"] = nil   --borramos entrada
  ```
- Claves y valores pueden ser de cualquier tipo
  ```
  t[print] = {}
  ```
  - Excepto `nil` como clave
- ```
  for k, v in pairs(t) do
    print(k, v)
  end
  ```

# Lua: tablas (2)

- `t.primero` es sinónimo de `t["primero"]`

```
local o = {}
o.pi = 3.14
o.circ = function(r) return 2*r*o.pi end
```

- Soporte especial para arrays (claves 1,2,3...n):
  - Pueden ser parte de una tabla cualquiera
  - Iterador de arrays `ipairs()`
    - `pairs()` recorre todo, no mantiene orden
    - `ipairs()` solo recorre la parte de array, en orden
  - Operadores especiales
    - `#` (longitud del array): agregar al final de un array es `t[#t+1]="a"`
    - `table.insert(), table.remove(), table.sort()`

# Lua: tablas (3)

```
function add(list, element)
    list[#list+1] = element
    list[element] = true
end

local my_list = {}

add(my_list, 'uno');
add(my_list, 'dos');
add(my_list, 'tres');

for k,v in ipairs(my_list) do
    print(k,v)
end

if my_list['dos'] then
    print (“El elemento 'dos' aparece!”)
end
```

# Lua: constructores de tablas

- `dias = {"martes", "miercoles"}`
  - `dias={}; dias[1]="martes"; dias[2]="miercoles"`
- `punto = {x=0, y=0}`
  - `punto={}; punto.x=0; punto.y=0`
- `a = {["es verdad"]=true,["mentira!"]=false}`
  - `a={}; a["es verdad"]=true; a["mentira!"]=false`
- 
```
triangulo = {
  {x=0,y=0},
  {x=0,y=1},
  {x=1,y=0},
}
```

# Lua: funciones

- Miembros de primer orden
  - Las funciones no tienen "nombre", tienen instancias en variables
- ```
  local printviejo = print
  print = function(...)
    printviejo(os.time(), ...)
  end
  print("ping!", "pong!")
  ```
  → 1302201902 ping! pong!
- ```
  local f = loadstring("b=4")
  f()
  print( b )
  ```
  → 4

```
local function f(x)
   return x
end
```

```
local f = function(x)
   return x
end
```

# Lua: funciones anónimas

```
local t= {"xs", "aaaa", "bab"}

table.sort(t, function(a, b) return #a<#b end)

for k,v in ipairs(t) do
   print (k,v)
end


-->

1  xs

2  bab

3  aaaa
```

# Lua: bibliotecas estándar

- Math
  - `math.cos`, `math.random`, `math.log`...
- Table
  - `table.insert`, `table.concat`, `table.sort`...
- String
  - `string.upper`, `string.match`, `string.gsub`...
- I/O
  - `io.open`, `io.write`, `io.read`...
- OS
  - `os.time`, `os.execute`, `os.getenv`...
- Debug

# Lua: otras bibliotecas

- Sistema
  - LuaPosix
  - lualibusb, librs232
  - LuaFileSystem
- Red
  - Básicos: LuaSocket, nixio
  - Aplicación: Kepler (plataforma), Xavante (webserver), Sputnik (CMS/wiki), Verse (XMPP)
- Texto
  - Lpeg, lrexlib
- Varios
  - LuaSQL, LuaSrcDiet, wxLua, LuaGTK...

# Lua: mis propias bibliotecas

- Como se hace un módulo:
  - mimodulo.lua:

```
local M = {}
local n=1 --privado al modulo
M.inc = function(x) return x+n end
return M
```

  - En el programa principal:

```
local m = require ("mimodulo")
print( m.inc(10) )
```

  - `Require` vs `dofile`

# Lua: Metatablas y Metamétodos

- Permite redefinir comportamiento de operadores
  - Principal uso: implementar estructuras de datos
  - Usar lo menos posible, pero no menos

```
function readOnly (t)
   local proxy = {}
   local mt = {        -- create metatable
        __index = t,
        __newindex = function (t,k,v)
            error("update on a r/o table!", 2)
        end
   }
   setmetatable(proxy, mt)
   return proxy
end
```

# Lua: Tablas *Weak*

- referencias a objetos que no impiden que el GC los recolecte.

```
local a={}
setmetatable(a, {__mode = "k"})
for i=1,3 do
    a[i]={'dato'..i}
    a[a[i]]=true
end
for k,v in pairs(a) do print("antes",k,v) end
table.remove(a,1)
collectgarbage()
for k,v in pairs(a) do print("despues",k,v) end
```

# Lua: Sandboxing

- Todo programa tiene una tabla asociada, que es su entorno: `_G`
- En el entorno están las variables globales
  - ...incluye las funciones estándar (!)
  - Por ejemplo, la variable global `print` está en `_G.print`, o lo que es lo mismo `_G["print"]`
- Se puede asignar una tabla cualquiera a una función para que sea su entorno:
  - `setfenv(f, {})`
- Se pueden armar entornos controlados para funciones no confiables: lo que no está en el entorno, no es accesible
  - `setfenv(f, {math=math, print=myprint})`

# Lua: Closures

```
function newCounter ()
  local i=0
  return function ()
           i=i+1
           return i
         end
end

local c1 = newCounter()
local c2 = newCounter()
print( c1() ) → 1
print( c1() ) → 2
print( c2() ) → 1
```

# Lua: iteradores (1)

```
function geb (n)
  return function ()
    if n % 2 == 0 then
      n = n/2
    else
      n = n*3+1
    end
    if n>1 then return n end
  end
end

for numero in geb(5) do
      print(numero)
end
```

# Lua: iteradores (2)

```
s="perez anda, gil camina."
hace = {}

for a, v in string.gfind(s, "(%a+)%s(%a+)") do
   hace[a] = v
end

print( "perez:", hace["perez"]  )
print( "gil:", hace["gil"]  )
```

# Lua: Corutinas

- Forma de multitarea cooperativa
- La única forma realmente multiplataforma (autocontenida)
  - Para plataformas específicas, existen módulos: LuaThread, LuaLanes, ConcurrentLua...
- `yield` es como un `return`, pero que no termina la función, solo la pausa. `resume` es como una llamada a una función, pero entra donde quedó pausada.
- Sirven para expresar máquinas de estado, pipelines, etc...

```
c = coroutine.create(function ()
                         print(1)
                         coroutine.yield()
                         print(2)
                     end)
coroutine.resume(c) --> 1
coroutine.resume(c) --> 2
```

# Recursión de cola

```
function estadoA ()
  print("estado A")
  if math.random()<0.5
    then return estadoA()
    else return estadoB()
  end
end

function estadoB ()
  print("estado B")
  if math.random()<0.5
    then return estadoA()
    else return estadoB()
  end
end

estadoA()
```

# Lua: Orientación a Objetos (1)

- Si, pero cual? (OO de C++ <> OO de Java <> OO de Python...)
- Tablas + funciones como miembros de primer orden es algo así como un objeto.
- Duck Typing: Lo que importa es que atributos tiene un objeto, no de que clase es.
- Se puede implementar herencia simple, múltiple, blá blá blá.
  - Recuerde, Lua no es Python, ni Java, ni C++...

# Lua: Orientación a Objetos (2)

- t.func(self, parametros) es t:func(parametros)
- Account = {
  withdraw = function (self, amount)
  self.balance = self.balance – amount
  end
  }

  local account = {
  balance = 0,
  withdraw = Account.withdraw,
  }

  Account.withdraw(account, 100.0)
  account:withdraw(100.0)

# Lua: Orientación a Objetos (3)

```
get_account = function ( initial )
   local balance = initial
   local account = {
      withdraw = function (amount)
         balance = balance – amount
      end
   }
   return account
end

local account = get_account(1000.0)

account.withdraw(100.0)
```

# Lua: Bindings

- Permite invocar módulos nativos desde Lua e invocar un script Lua desde un programa en C
- Basado en un stack:
  - Para leer parámetros, hacemos pop()
  - Para pasar parámetros, hacemos push()

```
static const struct luaL_reg libusb [] = {
   { "squareroot", l_sqrt },
}
static int l_sqrt (lua_State *L) {
    double n = luaL_checknumber(L, 1);
    lua_pushnumber(L, sqrt(n));
    return 1; /* number of results */
}
```

# Lua: SBCs y microcontroladores

- LuaJIT
  - x86/x64, ARMv5+ y ARM9E+, PPC/e500, MIPS
- LuaVM
  - Desde casi cualquier cosa, hasta algún Linux embebido sobre MIPSEL con 16MB de RAM y 8MB de flash
- pbLua
  - Para Lego NXT: ARM7, 64kB RAM, 256 kB flash
- eLua
  - x86, ARM7, ARM966E-S, Cortex-M3, AVR32...

# Lua: IDE

- Lenguaje débilmente tipado: los IDEs no ayudan tanto como suelen hacerlo
  - LuaEclipse, SciTE, TextAdept, gedit, ZeroBraneStudio...
  - LuaInspect
  - Documente parámetros y estructuras!
- Debugging
  - `assert()`
  - `print()`
  - strict.lua: controla acceso a variables globales
  - http://lua-users.org/wiki/DebuggingLuaCode
- Profiling
  - LuaProfiler
  - dotlua.lua: visualiza el estado de la aplicación.

# Lua: cuidado con.

- Variables globales por defecto
  - Usar `local` siempre que se pueda
  - `require ("strict")`
- Arrays
  - Se numeran a partir de 1
  - Cuando hay "agujeros", `#` no esta bien definida. Si necesitamos marcar ausencia, usar un elemento nulo explícito (p.ej. `local nada={}`)
- Strings inmutables
  - Cuidado con las concatenaciones largas en un bucle...
  - Usar `table.concat()`
- Boolean
  - Falsos solo `false` y `nil` (`0 es verdadero!`).

# Referencias

Presentaciones:
http://www.inf.puc-rio.br/~roberto/talks/

Programming in Lua:
http://www.lua.org/pil/