Functions in Lua

Р. Иерусалимский
What does “function” mean?
It means several things...
Functions are First-Class Values

• Functions are values.
  – or, there are values that represent functions.
• These values can be stored in variables and data structures.
• They can be passed as arguments to and returned by other functions (*higher-order functions*).
• They can be called anywhere in a program.
Functions can be Nested

• We can define functions inside other functions.
  – recursively

```python
function foo (x)
    function p (y)
        print(y)
    end
    p(2*x)
end
```
There are Anonymous Functions

- We can write a function without giving a name to it.
- Syntactically, we can write a function as an expression in the language.

```add = (function (x,y) return x+y end)```
Nested Functions have Lexical Scoping

- A function can access local variables from its enclosing functions.
- A function can escape from its enclosing function (e.g., by being returned) and still access those variables.

```plaintext
function makecounter (n)
    return function (d)
        n = n + d
        return n
    end
end

c = makecounter(10)
print(c(1))   --> 11
print(c(3))   --> 14
```
Properties Somewhat Independent

- C has functions as first-class values, but no nesting.
- Lisp (original) has functions as first-class values and anonymous functions, but no lexical scoping.
- Pascal has lexical scoping, but functions are not first-class values.
- Python 2 and Java have lexical scoping, but only for values.
- *Blocks* in Ruby and Smalltalk are anonymous with lexical scoping, but they are not first-class values.
How Lua uses functions to achieve its goals
What are the 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.
  – FreeBSD, 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)
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.
How Lua uses functions to achieve its goals
Modules

- Tables populated with functions

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

local f = math.sqrt
print(f(10))
```

- 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

• Lexical scoping (for local definitions)
• Pros
  – needs no new features
  – easy to interface with other languages
  – flexible
• Cons
  – not as good as “the real thing” (regarding syntax)
  – too dynamic (?)
Eval

- Hallmark of dynamic languages.
- Lua offers a “compile” function instead.

```lua
function eval (code)
    -- compiles source 'code' and
    -- executes the result
    return \(\text{load}(\text{code})(())\)
end

function load (code)
    -- creates an anonymous function
    -- with the given body
    return \(\text{eval}(\text{return function () " .. code .. " end"})\)
end
```
Load

- Clearly separates compilation from execution.
- `load` is a pure function.
- It is easier to do `eval` from `load` than the reverse.
- Any code always runs inside some function.
  - we can declare local variables, which naturally work like static variables for the functions inside the chunk.
  - chunks can return values.
Exception Handling

- All done through two functions, pcall and error

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

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

• Anonymous functions with lexical scoping

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

• Cons
  – verbose
  – body cannot return/break
  – try is not cost-free
Iterators

• Old style:

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

- Anonymous functions (for old style), lexical scoping

- Pros
  - easy to interface with other languages

- Cons
  - cannot traverse nil
  - not so simple as explained
Objects

- first-class functions + tables $\approx$ objects
- syntactical sugar for methods
  - handles self

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

a:foo(x)  $\rightarrow$  a.foo(a,x)

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

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

• Cons
  – may need some work to get started (DIY)
  – no standard model (DIY)
The Lua-C API

- Functions are constructs found in most languages, which compatible basic semantics.
- Constructions based on functions are easier to translate between different languages.
- Modules, OO programming, and iterators need no extra features in the Lua-C API.
  - all done with standard mechanisms for tables and functions.
- Exception handling and `load` go the opposite way: primitives in the API, exported to Lua.
Implementation

- Based on *closures*.

- A closure represents the code of a function plus the environment where the function was defined.

- Lua uses *upvalues* to represent the environment, one for each external variable used by the function.

- Zero cost when not used.
  - variables live on the stack.
Basic data structures

variable in the stack

a) open upvalue

b) closed upvalue
List of open upvalues (for unicity)
Closing an upvalue
Several Details...

- One-pass compiler.
- Safe for space.
- Uses flattening for nesting.
- List of open upvalues is limited by program syntax.
- A closure may point to upvalues in different stacks.
Final Remarks

• Lua is not only about tables.
• Like with tables, Lua itself uses functions for several important constructs in the language.
• In Lua, the use of constructors based on first-class functions greatly helps to make the C API general.