

# The History of Lua

Tero Hasu

`tero.hasu@hut.fi`

Helsinki University of Technology

T-106.5800 Seminar on Software Techniques  
Seminar on the History of Programming Languages  
17 November 2009

# Outline

## 1 Lua Introduction

- Lua Overview
- Lua Influences

## 2 Lua History

- Lua Origins
- Lua Evolution
- Lua Today

## 3 Lua Features

# Lua Language Overview

- “scripting language”
- relatively simple
- dynamically typed
- distinguishing features: *tables* (associative arrays), extensible semantics, coroutines
- typically embedded into applications
  - often as a configuration language

# Syntax Taster

```
-- Recursive impl.
function fact(n)
  if n == 0 then
    return 1
  else
    return n * fact(n-1)
  end
end

-- Iterative impl.
function fact(n)
  local a = 1
  for i = 1, n do
    a = a * i
  end
  return a
end
```

# Lua Implementation

- available from [www.lua.org](http://www.lua.org) (MIT license)
  - open source, but "closed development"
- smallish, 17000 lines of C
- portable
- embeddable
  - can call Lua from C and C from Lua
- clean code
  - good for your “code reading club”, reading advice at <http://bit.ly/4ynQDr>

# Lua Portability

- written in "clean C", i.e. an intersection of C and C++ (and Objective-C)
- no library dependencies to speak of
  - no Boehm GC library, for instance (a common problem with otherwise portable language implementations like ooc<sup>1</sup> and Shed Skin<sup>2</sup>)
- Personal experience<sup>3</sup>: Easier to port to Symbian OS than e.g. SQLite or iksemel.
  - just edit the config header, basically

---

<sup>1</sup>[www.ooc-lang.org](http://www.ooc-lang.org)

<sup>2</sup>[code.google.com/p/shedskin/](http://code.google.com/p/shedskin/)

<sup>3</sup>[www.contextlogger.org](http://www.contextlogger.org)

# Lua Performance

- fast for an interpreted scripting language
  - language simplicity helps
- one-pass compiler (emits VM instructions as it parses)
  - fast to compile, but one-pass nature rules out some optimizations in generated code
- lexer and parser hand written for performance
- presently has a register based VM
  - potentially faster than stack based, but portability rules out compiler-specific VM optimizations
- pre-compilation supported

# Lua Adoption

- Lua is fairly popular
  - particularly in game development
    - *the* most popular game scripting language (according to some surveys)
    - decent performance, not resource hungry, easy to embed (and hence deploy), suitable for data files, portable to non-conventional platforms (e.g., PS3, iPhone)
  - better-known software using Lua: Adobe Photoshop Lightroom, lighttpd, **LuaTeX**, Monotone, Nmap, sio2, VLC, WoW, ...

# Lua Community

- breakthrough exposure in 1996 (around Lua v2.4–2.5)
  - articles in Software: Practice & Experience and Dr. Dobb's Journal
- mailing list created in February 1997
  - focal point of the Lua community
- LuaForge<sup>4</sup> hosting around 500 projects

---

<sup>4</sup>luaforge.net

# Lua Influences

Lua was designed for production use, not as a research vehicle. Initially, no new concepts, apart from its data-description facilities.

- **Modula**
- **Scheme**
- Multiple assignment and multiple return values from function calls were taken from **CLU**.
- The idea of allowing a local variable to be declared only where needed was taken from **C++**.
- Associative arrays (called “tables” in Lua) were taken from **SNOBOL** and **Awk**.

## Lua vs Modula Syntax

- “Syntactically, Lua is reminiscent of Modula and uses familiar keywords.” [HOPL]

```
-- Lua.
function fact(n)

  local a = 1

  for i = 1, n do
    a = a * i
  end
  return a
end
```

```
-- Modula-2.
PROCEDURE Fact(n: CARDINAL):
  CARDINAL;
VAR a: CARDINAL;
BEGIN
  a := 1
  FOR i := 1 TO n DO
    a := a * i;
  END;
  RETURN a;
END Fact;
```

## Scheme—A Source of Inspiration

- “The influence of Scheme has gradually increased during Lua’s evolution.” [HOPL]

### Greenspun’s Tenth Rule of Programming

*Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp.*

## Scheme and Lua Similarities

- Dynamic typing, first-class values, anonymous functions, closures, ...
  - would have wanted first-class continuations
- `function foo() ... end`  
`foo = function () ... end`  
  
`(define (foo) ...)`  
`(define foo (lambda () ...))`
- Scheme has lists as its data structuring mechanism, while Lua has tables.
- No particular object or class model forced onto the programmer—choose or implement one yourself.

# Tecgraf

- Tecgraf (Computer Graphics Technology Group of PUC-Rio<sup>5</sup> in Brazil)
  - Lua creators: Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes
  - industrial partner: Petrobras (an oil company)
- strong trade barriers in Brazil (1977–1992)
  - political, financial, and bureaucratic problems with buying customized software from abroad

---

<sup>5</sup>Pontifícia Universidade Católica do Rio de Janeiro

# Lua Ancestors

- “These languages, called DEL and SOL, were the ancestors of Lua.” [HOPL]
- DEL and SOL were domain-specific languages (DSLs) for Tecgraf-developed interactive graphical programs
  - programs used at Petrobras for engineering tasks

# DEL

- DEL as in "data-entry language"
  - for describing data-entry tasks—for specifying:
    - named and typed fields
    - data validation rules
    - how to input and output data
- “DEL did not influence Lua as a language.” [HOPL]
  - The main influence was “the realization that large parts of complex applications could be written using embeddable scripting languages”.

# SOL

- SOL as in "Simple Object Language"
  - a DSL for a configurable report generator for lithology<sup>6</sup> profiles
- SOL interpreter: read a report description, and syntax and type check specified objects and attributes
- syntax influenced by BibTeX

```
type @track{ x:number, y:number=23, id=0 }
T = @track{ y=9, x=10, id="1992-34" }
```

---

<sup>6</sup>the study of rocks

## Motivation for Lua

- DEL users began to ask for more power, e.g. control flow (with conditionals and loops)
- SOL implementation finished, but not delivered, as support for procedural programming was soon to be required
- conclusion: replace both SOL and DEL by a single, more powerful language

## Existing Alternatives

- Tcl: "unfamiliar syntax", bad data description support, Unix only
- Lisps: "unfriendly syntax"
- Python: still in its infancy

No match for the free, do-it-yourself atmosphere at Tecgraf.

## Birth of Lua

- "Lua"—"moon" in Portuguese
  - cf. "SOL"—"sun" in Portuguese
- SOL's syntax for record and list construction

```
T = @track{ y=9, x=10, id="1992-34" }
```

- Semantics differ:
  - tables represent both records and lists;
  - `track` (here) does not name a record type, it names a function to be applied.

# Lua 1

- Lua 1.0 finished in July 1993.
  - No public release.
  - Version number given after the fact.
  - "The simplest thing that could possibly work."
    - `lex`, `yacc`, a stack-based VM, a C API, 5 built-in functions, 3 small libraries
  - A success in Tecgraf projects.
- Lua 1.1 in July 1994.
  - released to the public, but under a restrictive license
  - bytecode compilation speedups (e.g., a hand-written lexer) to support large graphics metafiles

## Lua 2

- OO hype had peaked in the early 1990s
  - user pressure for object-oriented Lua
- Lua 2.1 in February 1995
  - introduced *extensible semantics*
    - designed to *allow* OO programming
- Lua 2.2 in November 1995
  - introduced a debug API
    - means for writing (in C) introspective tools (such as debuggers and profilers)
    - access to e.g. call stack and currently executing line

## Lua 2

- Lua 2.4 introduced `luac`, a separate bytecode precompiler
  - avoid costly on-the-fly parsing and code generation
    - reduce Lua footprint by removing lexer, parser, and code generator
  - get "off-line" syntax checking by running through `luac`
  - prevent casual reading and modification of code
- `strfind` and `gsub` added in v2.5
  - authors wanted pattern matching
    - third-party engines tend to be large (e.g., many PL implementations today use the PCRE library)
  - basic patterns, not full regexps
  - innovation: `gsub` alternatively accepted a function to specify a replacement

## gsub

```
> x = string.gsub("abbc", "b", "B"); return x  
aBBc
```

```
> x = string.gsub("abbc", "b",  
  function (x) return "B" end);  
  return x  
aBBc
```

```
> x = string.gsub("abbc", "b", string.upper);  
  return x  
aBBc
```

# Lua 3

- Lua 3.1 (July 1998)
  - introduced anonymous functions and function closures via "upvalues"
    - motivated by higher-order functions such as `gsub`
    - full lexical scoping not until v5.0
  - support for multiple independent Lua states, switchable at runtime
  - added variants for many VM instructions—too complicated for little performance gain, so removed in v3.2
- Lua 3.2 (July 1999)
  - debug API with a Lua interface

## Lua 4

- Lua 4.0 in November 2000
  - C API redesigned, and more complete
    - not backward compatible
    - design based on a stack metaphor for exchanging values with C
    - fully reentrant API, with Lua state specifiable explicitly
    - no more built-in functions—all standard libraries on top of the C API
  - introduced a `for` statement
    - previously only `while` loops
    - Lua v3.1 had introduced `foreach` higher-order function
- Lua 4.1 final never released

## Lua 5.0

- released in December 2003
- full lexical scoping
- *threads*, a basic mechanism of multiple stacks
  - also: external multithreading and coroutines
- library functions packaged in tables
  - `strfind` → `string.find`
- register-based VM
  - the first register-based virtual machine to have wide use(?)
  - does better in computationally intensive benchmarks
  - fewer moves, fewer instructions, but more complex compilation and decoding
    - Shi et al. Virtual Machine Showdown: Stack Versus Registers. 2008.

# Lua 5.1

- released in February 2006
- incremental GC
  - requested by game developers, to avoid long pauses
- module system
  - policies for writing modules, and tweaks to better support them

# Lua Feature Evolution

- Lua designers have shown good judgement.
- Learn **PL design** by asking:
  - What features were added to Lua and why?
  - What features were *turned down* and why?
- Learn **PL implementation** by asking:
  - How were the features implemented?
  - What kind of implementations were *not* possible due to other implementation choices?

# Lua Types

Lua's type selection has remained fairly stable.

- initially: numbers, strings, tables, nil, userdata (pointers to C objects), Lua functions, C functions
- unified functions in v3.0
- booleans and threads in v5.0

# Tables

- any value as index—can say present a set by storing its elements as table indices
- early syntax: `@()`, `@ [1, 2]`, `@ {x=1, y=2}`
- later syntax: `{}`, `{1, 2}`, `{x=1, y=2}`, `{1, 2, x=1, y=2}`
  - sparse arrays OK: `{ [1000000000] = 1 }`
- `f oo { ... }` is sugar for `f oo ( { ... } )`
- element referencing sugar: `a . x` for `a [ "x" ]`
  - as in FAS, array indexing is from 1
- tables with named functions for OO
  - for inheritance, define a table indexing operation

## Extensible Semantics

- *fallbacks* in Lua 2.1
  - one function per operation (table indexing, arithmetic operations, ...), if operation not otherwise applicable to a value
- *tag methods* in Lua 3.0
  - tag-specific fallbacks, any value taggable
- *metatables* and *metamethods* in Lua 5.0

```
x = {}
function f () return -5 end
setmetatable(x, { __unm = f })
return -x      --> -5
```

## Closures via "Upvalues"

- Lua authors wanted lexical scoping early on
  - difficult due to technical restrictions
    - wanted to keep a simple array stack for activation records
    - one-pass compiler
- Lua 3.1 with a compromise called *upvalues*
  - In creating a function, make (frozen) copies of the values of any external variables used by a function.

```
function f ()  
  local b = 1  
  return (function () return %b + 1 end)  
end  
return f() () --> 2
```

## Full Lexical Scoping

- Lua 5.0 got the real thing
- Solution: "Keep local variables in the (array-based) stack and only move them to the heap if they go out of scope while being referred by nested functions." (JUCS 11 #7)

```
function f ()  
  local b = 1  
  local inc_b = (function () b = b + 1 end)  
  inc_b()  
  return (function () return b end)  
end  
return f() () --> 2
```

# Tail Calls

- tail calls supported since 5.0
  - called function reuses the stack entry of the calling function
    - erases information from stack traces
- only for statements of the form `return f(...)`
  - `return n * fact(n-1)` does *not* result in a tail call

# Coroutines

- *coroutines*—a general control abstraction
  - term introduced by Melvin Conway in 1963
  - has lacked a precise definition, but implies "the capability of keeping state between successive calls" [de Moura]
- have not been popular in mainstream languages
- classification:
  - *full coroutines* are stackful, and first-class objects [de Moura]
    - *stackful coroutines* can suspend their execution from within nested functions
  - an *asymmetric coroutine* is "subordinate" to its caller—can yield, caller can resume

## Coroutines in Lua

- constraints: portability and C integration
  - cannot manipulate a C call stack in ANSI C
  - impossible: first-class continuations (as in Scheme), symmetric coroutines (e.g., in Modula-2)
- Lua 5.0 got *full asymmetric coroutines*, with `create`, `resume` and `yield` operations
  - ...and PUC-Rio guys gave proof of ample expressive power in [de Moura]
  - capture only a partial continuation, from `yield` to `resume`—cannot have C parts there

## Coroutine Example

```
> return (string.gsub("abbc", "b",  
    function (x) return "B" end))  
aBBc
```

```
> return (string.gsub("abbc", "b",  
    coroutine.wrap(function (x)  
        coroutine.yield("B")  
        coroutine.yield("C")  
    end)))  
aBCc
```

## C API

```
function foo(t) return t.x end
```

```
void foo_l(void) { /* Lua 1.0 */
  lua_Object t = lua_getparam(1);
  lua_Object r = lua_getfield(t, "x");
  lua_pushobject(r);
}
```

```
int foo_l(lua_State* L) { /* Lua 4.0 */
  lua_pushstring(L, "x");
  lua_gettable(L, 1);
  return 1;
}
```

## Lua in Retrospect

- worked well:
  - tables as the sole data structuring mechanism (uniformity, flexibility)
  - portability (increased adoption)
  - design by small committee: listen to users, but only add a feature when *all* three designers in agreement
- no regrets:
  - simplicity and elegance over backward compatibility
- regrettable:
  - no boolean type until v5.0 (could have reserved `nil` as "no useful value")
  - implicit coercion of strings to numbers in arithmetic operations (taken from Awk, troublesome)
  - no module system until v5.1 (had discouraged code sharing)

## Future of Lua

- LuaJIT v2.0 beta is out
  - JVM-level performance, but x86 only for now
- Provide Lua with extensible syntax to go with extensible semantics?
  - Lua v3 already had conditional compilation
  - a full macro processor has been the most requested feature
    - authors would like Lua programmable macros
  - there is Metalua, a Lua 5.1 fork with a macro system
- Threatened by JavaScript(?)
  - blending of web and native technology in applications increasing in popularity
  - an interpreter may already be there
    - e.g. Qt 4.3-up comes with ECMAScript

## References

**[HOPL]** Ierusalimschy et al. The Evolution of Lua. HOPL-III (2007).

**[Lua5]** Ierusalimschy et al. The Implementation of Lua 5.0. Journal of Universal Computer Science 11 #7 (2005).

**[de Moura]** de Moura and Ierusalimschy. Revisiting Coroutines. ACM TOPLAS 31 #2 (2009).