Lua Programming Language

An Introduction

Dr. Christian Storm
TNG Big Techday 7
23rd May 2014
About Lua

- invented as configuration and data description language
- first version released 1993, current version is 5.2.3
- Lua is interpreted, dynamically typed, garbage collected, has closures, coroutines, <insert fancy stuff here>, …
- Lua is
  - clean & simple: a designed, not evolved language
  - fast: even faster with LuaJIT
    - see, e.g., Computer Language Benchmarks Game, Hash benchmark
  - small: liblua.so.5.2.3 is 200K, 60 source files, 14,728 lines code (C,C++,make)
  - portable: written in ANSI C/C++
  - embeddable & extensible: C/C++, Java, C#, Perl, Python, Ruby, …
- Lua complements C’s low level power (e.g., via inline Assembler)
  - high(er) level language expressibility without having to use C++ 😊
Lua … so what?

▶ empowers your Texas Instruments EK-LM3S to play Pong
Lua … so what?

... and Snake
Lua … so what?

... and even Tetris!
Lua … so what?

- scripting the UI subsystem of Blizzard’s World of Warcraft
Lua … so what?

- scripting Crytek’s CryEngine-based games, e.g., FarCry and Crysis
Lua … so what?

Lua Programming Language – An Introduction

almost everywhere in Adobe’s Photoshop Lightroom
Lua … so what?

- awesome X11 window manager
  [link](http://awesome.naquadah.org)

- vim editor
  [link](http://www.vim.org)

- VLC Media Player
  [link](http://www.videolan.org/vlc/)

- LuaTEX
  [link](http://www.luatex.org)

- Angry Birds
  [link](http://www.angrybirds.com)

- Nginx HTTP Server
  [link](http://nginx.org)

- Wireshark
  [link](http://www.wireshark.org)

- NetBSD's Kernel
  [link](http://www.netbsd.org)

- Havok Engine
  [link](http://www.havok.com)

▶ and in may other places you probably wouldn’t expect it …
Outline

Lua Language Basics

- Syntax
- Data Types
- Statements and Control Structures
- Functions
- Closures

More Advanced Lua

- Modules
- Coroutines
- Metatables and Metamethods
- OOP in Lua
What does Lua look like?

```lua
function factorial(n, ans)
    ans = ans and ans or 1
    if ans == math.huge then
        print("E: overflow")
        return nil
    end
    if n ~= 0 then
        return factorial(n-1, n*ans)
    end
    return ans
end

fact = factorial(arg[1] and tonumber(arg[1]) or 0)
print(fact)

> lua propertailrecursionfactorial.lua 5
120
> _
Lua’s Syntax

chunk ::= block
block ::= {stat} [retstat]
stat ::= ';' | varlist '=' explist | functioncall | label | break | goto Name | do block end |
  while exp do block end | repeat block until exp |
  if exp then block {elseif exp then block} [else block] end |
  for Name '=' exp ',' exp [';',']' exp] do block end |
  for namelist in explist do block end | function funcname funcbody |
  local function Name funcbody | local namelist ['=' explist]
retstat ::= return [explist] [';']
label ::= '::' Name '::'
funcname ::= Name {'.'} Name [':'] Name
varlist ::= var {',', var}
var ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name
namelist ::= Name {',', Name}
explist ::= exp {',', exp}
exp ::= nil | false | true | Number | String | '...' | functiondef | prefixexp | tableconstructor |
  exp binop exp | unop exp
prefixexp ::= var | functioncall | '(' exp ')
functioncall ::= prefixexp args | prefixexp '.' Name args
args ::= '(' [explist] ')') | tableconstructor | String
functiondef ::= function funcbody
funcbody ::= '(' [parlist] ')' block end
parlist ::= namelist [',', '...'] | '

unop ::= '-' | not | '#'

binop ::= '+' | '-' | '*-' | '/' | '^' | '%' | '..' | '<' | '<=' | '>' | '>=' | '==' | '~=' | and | or (**)

(*/) operator precedence is missing

http://www.lua.org/manual/5.2/manual.html#9
Lua Language Basics – Syntax

... compared to Python 3.4’s Syntax

```python
try_stmt: ('try' ': suite (except_clause ': suite)+ ['else' ': suite]
  ['finally' ': suite] | 'finally' ': suite]) suite
with_stmt: 'with' suite [with_item (', ', with_item) ': suite
with_item: test (test as_expr)
except_clause: 'except' test [as_expr]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
test: or_test | 'if' or_test 'else' test | lambda
test_nocond: or_test | 'lambda' varargslist ':' test
lambda_nocond: 'lambda' varargslist test_nocond
or_test: test | and_test
and_test: test | not_test
not_test: not_test 'or' and_test
not: not_test 'not' | test_not
expression: expr
test: expression
not_test: test
and_expr: test 'and'
test: and_expr
or_expr: test 'or'
test: or_expr
xor_expr: test 'xor'
test: xor_expr
and_test: test 'and'
test: and_test
or_test: test 'or'
test: or_test
comparison: expr (comparison '==' | '!=' | '>' | '<' | '>=' | '<=' | '!=' | '==' | '<>'
<>
term: factor (('+'|'-') term)*
term: factor ('('*'|'/'|'%'|'//') factor)*
factor: (('*'|'+'|'-'|'&'|'|'|'^' factor)
expression: '(' test ')'
test: expression
atom: (''' [yield_expr|testlist_comp] ')'
  ['[ testlist_comp ]']
['[dictorsetmaker]']
NAME | NUMBER | STRING | ... | 'None' | 'True' | 'False'
testlist: test (testlist ',' test)*
testlist_comp: test|comparision|test
lambda: (arglist ':' [test|star_expr])
arglist: (',' var_arglist)*
var_arglist: varargslist
star_expr: test
arithmetic: (testlist_comp) atom
atom: (''' [yield_expr|testlist_comp] ')'
  ['[ testlist_comp ]']
['[dictorsetmaker]']
NAME | NUMBER | STRING | ... | 'None' | 'True' | 'False'
testlist: test (testlist ',' test)*
testlist_comp: test|comparision|test
lambda: (arglist ':' [test|star_expr])
arglist: (',' var_arglist)*
var_arglist: varargslist
star_expr: test
arithmetic: (testlist_comp) atom
atom: (''' [yield_expr|testlist_comp] ')'
  ['[ testlist_comp ]']
['[dictorsetmaker]']
NAME | NUMBER | STRING | ... | 'None' | 'True' | 'False'

https://docs.python.org/3/reference/grammar.html

Lua Programming Language – An Introduction
Basic Data Types

- **nil**
  - *nil* is the “nothing” value (cf. *null* in C)
  - *nil* is the “value” of undefined variables

  ```lua
  print(a)  --> nil
  a = 42
  a = nil   -- a is no longer ”existing”
  ```

- **boolean**
  - ordinary boolean values *true* and *false*
  - only *nil* and *false* are “false”, all others are “true”

  ```lua
  a = 0     -- a evaluates to true in condition
  a = nil   -- a evaluates to false in condition
  ```

- **userdata**
  - *void* pointers to C data structures
  - C data stored in Lua variables
  - sharing of non-primitive data types among C and Lua, e.g., *struct* data
Basic Data Types

- numbers
  - underlying numerical data type is 64 Bit double precision floating point
  - follows IEEE 754, i.e., no rounding problems for integers up to $2^{53}$
  - 64 Bit integer data type proposed for Lua 5.3

  ```
  a = 23
  a = 5.0
  a = 12/5
  a = 1.5e+2
  a = 0xCafe
  ```

- strings
  - sequence of characters, garbage collected
  - eight-bit clean, i.e., may contain any characters including numeric codes
  - are immutable (memoized) as used in table access as key

  ```
  a = 'cat'
  a = "dog"
  a = "cat" .. 'dog' -- creates new (memoized) concatenated string "catdog"
  ```
For Convenience: Coercion

- automatic type conversion between string and number at run-time
  - arithmetic operation on a string tries to convert it to a number
  - string operation on a number tries to convert it to a string
- explicit conversions available: tonumber(), tostring()

```lua
print("10" + 1)  -- > 11
print(10 .. ""  == "10")  -- > true
print("hello" + 1)  --> error, cannot convert "hello" to number
print(tostring(10) == "10")  -- > true
```
The Table Data Type

- sole and omnipresent advanced data type in Lua
- associative array, i.e., a key=value store
- anonymous, no fixed relation between table and variable holding it
- statements manipulate references (pointer) to a table

```
a = {}                     -- create empty table and bind it to a
a["foo"] = "bar"          -- assign the value bar to the key foo
a[123] = 456               -- assign the value 456 to the key 123
a = {"foo"="bar", [123]=456} -- same effect as above three statements

print(a["foo"])            -- bar
key="foo"; print(a[key])    -- bar
print(a["(!)"])            -- nil -- non-existent keys have default “value” nil

b = a                       -- b points to same table as a
a = nil                     -- (anonymous) table still referenced by b
b = nil                     -- table is unreferenced, garbage collected on next cycle

-- shortcut syntax for string keys following [_a-zA-Z][_a-zA-Z0-9]*
a = {foo="bar"}            -- same as a = {"foo" = "bar"}
a.foo = 123                 -- same as a["foo"] = 123
```
The Table Data Type

- special case: contiguous integer keys 1, 2, ... form an “array”-like

```lua

print(a[0]) --> nil -- 1-indexed as it’s just a key, not an offset [得意]
print(a[1]) --> a

print(#a) --> 3 -- "length", i.e., number of *contiguous* integer keys [得意]
a[2] = nil -- a[2] is a "hole" now
print(#a) --> 1 -- can be undefined, don’t use # with sparse array tables!

a = {"a", "b", "c"}
table.remove(a, 2) -- sets a[2] = nil *and* shifts down any integer keys >2
print(#a) --> 2 -- this looks better!

a = {"a", "b", "c"}
a["foo"] = "bar" -- assign the value bar to the string key foo
a[#a+1] = "d" -- append value d, i.e., a[4] = "d"
print(#a) --> 4 -- a has 4 contiguous integer keys (and one string key)
```
Statements and Control Structures

- **do ... end**
  - explicitly defines a block (and a scope)

- `<variable1>[,<variable2>,...] = <value1>[,<value2>,...]`
  - defines a global variable (in the globals table \_G[<variable>] = <value>)
  - variable declaration is a statement
    - effective only after execution of the statement
    - declarable where necessary, not bound to particular position or block scoping
  - by default, variables are global (unlike, e.g., Python) [ squarely]

- **local** `<variable1>[,<variable2>,...] [= <value1>[,<value2>,...] ]`
  - defines a variable local to a block (and its inner blocks)
  - scope ends on block’s last non-void statement
  - may shadow same-named global or local variable from outer blocks [ squarely]
Statements and Control Structures

- `::<label>::` and `goto <label>`
  - a more powerful `continue`-alike, not Dijkstra’s considered harmful goto 😞
  - `::<label>::` and `goto <label>` must be in the exact same block/scope
  - hence no `goto` jump into another block, out of a function, ...

- `if <condition> then <block>`  
  `elseif <condition> then <block>`  
  `else <block> end`

- `while <condition> do <block> end`

- `repeat <block> until <condition>`
  - the scope of `<block>`’s local variables extends to `<condition>`

- `for ctr=cstart,cend[,cinc] do <block> end`
  - `cstart`, `cend`, `cinc` are evaluated once before loop starts
  - `ctr` is automatically created local variable
Statements and Control Structures

- **for** `a1,a2,... in <iterator>() do <block> end`
  - `a1,a2,...` are automatically created local variables
  - `iterator pairs(table) → key, value` loop over all key=value pairs in no particular order
  - `iterator ipairs(table) → index, value` ordered loop over all integer keys 1, 2, ... until the first `nil`

- **break**

- **return** `<value1> [, <value2>, ...]`
  - `return` must be the last statement of a block for syntactic reasons, i.e., before `end, else, elseif, or until`
  - `do return <value1> [, <value2>, ...] end` “circumvents” this restriction
  - implicit return at end of a function
Functions

- Functions are first-class values, they can be stored in
  - Local and global variables
  - Table keys and values
- First-class functions + tables ≈ “objects”

```lua
function f(param)
    local param = param or 1  -- set local variable param = 1 if param == nil
    print(param)
end
-- function f(param) ... end ⇔ f = function(param) ... end

local g = f
f("Example") --> Example
g("Example") --> Example

function varargs(...)  -- '{...}' is an "array"-table of parameters
    for _, v in ipairs({...}) do print(v) end
end
varargs(1) --> 1
```
Closures

- lexical scoping: a function’s full access to its enclosing local variables, in Lua speech: upvalue
- closure: “function plus all it needs to access upvalues correctly”

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

a = newCounter()
print(a()) --> 1
print(a()) --> 2

b = newCounter()  -- new closure, new upvalue variable i
print(b()) --> 1
print(b()) --> 2
print(b()) --> 3
```
Outline

Lua Language Basics
- Syntax
- Data Types
- Statements and Control Structures
- Functions
- Closures

More Advanced Lua
- Modules
- Coroutines
- Metatables and Metamethods
- OOP in Lua
More Advanced Lua – Modules

(1/2)

Modules

- modules function as namespaces and structuring mechanism
- a module is some code chunk returning a table of exports (per convention)
  - code chunk mod.{lua,so} is “sourced” into current scope by `require("mod")`
  - a returned table of exports is “cached” as `package.loaded["mod"]`
  - further `require("mod")` calls return `package.loaded["mod"]`
  - reload a module via `package.loaded["mod"] = nil; require("mod")`
- modules are first-class values – as tables are
Modules Example

- **example module mod.lua**

  ```lua
  local mod = {}  -- public interface table
  local function _div(a,b) return a/b end
  function mod.div(a,b) return _div(a,b) end
  mod.attr = 23  -- exported module function div()
  globvar = 96   -- exported module variable attr
  return mod
  ```

- **usage of mod.lua**

  ```lua
  local mod = require("mod")  -- load mod.lua & bind mod to package.loaded["mod"]
  local div = mod.div           -- bind div to function mod["div"]
  print(div(84,2)) --> 42
  print(mod.attr) --> 23
  print(globvar) --> 96  -- as globvar is "sourced" into _G["globvar"]
  ```
Collaborative Multitasking with Coroutines

- coroutine: a function that may yield anytime and be resumed later
  - only one coroutine runs at a time, i.e., cooperative scheduling
  - suspends its execution deliberately via `yield()`, never preemptively

- caller and coroutine can exchange data via `coroutine.resume()` and `coroutine.yield()`

- used for producer/consumer-like state-based patterns, e.g., generators:

  ```lua
  function generator(a, b)
    -- wrap()-returned function implicitly calls coroutine.resume() when called
    return coroutine.wrap(function()
        for n = a, b do coroutine.yield(n) end
    end)
  end

  for item in generator(1, 5) do print(item) end
  ```
Metatables and Metamethods

- A metatable is a table consisting of metamethods.
- Metamethods define or override the behavior of a type or value, cf. Python’s `__add__()`, `__getattr__()`, `__setattr__()`, ...
- Definable metamethods are, e.g.,
  ```lua
  __add__(a, b)  -- addition of two values: a + b
  __index__(a, b)  -- table indexing access: a[b]
  __call__(a, ...)  -- when calling a value: a(...)  
  ...
  ```
- Every type has an associated default metatable.
- A value’s metatable defaults to its type’s metatable.
- Only table metamethods are overridable from within Lua, use C for others.
- Used to implement “classes”, inheritance, to overload operators, …
Metatable and Metamethod Example

```lua
a = { value = 1 }
b = { value = 2 }

print(a+b)  -- attempt to perform arithmetic on global 'a' (a table value)
            -- FAILS since <table> + <table> is not defined

-- so, define ametatable with a metamethod defining a + b
addmt = {
    __add = function(a,b)
        return a.value + b.value
    end
}
setmetatable(a, addmt)

print(a+b)  -- 3
```
Introductory OOP Example

Prototype = {
    attribute = ”attribute value”,
    method = function(self) print(self.attribute) end,
    new = function(self, object)
        object = object or {} -- set or create initial object table
        self.__index = self -- set object’s lookup to Prototype
        return setmetatable(object, self) -- return newly created object (table)
    end
}
PrototypeMT = {
    __call = function(self, ...) return self.new(self, ...) end
}
setmetatable(Prototype, PrototypeMT )

obj1 = Prototype()
obj1:method() --> attribute value

function obj1:method() print(self.attribute, ”[override]”) end
obj1:method() --> attribute value [override]

Note: using some syntactic sugar for brevity of presentation

▶ more sophisticated examples can be found in, e.g., MiddleClass and Classy
**Conclusion**

- pick the right language for the problem at hand
  … and now this might be Lua 😊
- less libraries and bindings than the “Big Ones”, e.g., Python, Perl
  but: bindings are easy and e.g. Penlight gives you batteries
- small code base, easy and fun to experiment with, e.g.,
  write your own module, memory allocator, garbage collector, …
- extensible and embeddable
- clear and expressive syntax
- simple but still powerful constructs
Thank You

Questions

✉️ christian.storm@tngtech.com