Lua Programming Language

An Introduction

Dr. Christian Storm

TNG Big Techday 7
23rd May 2014

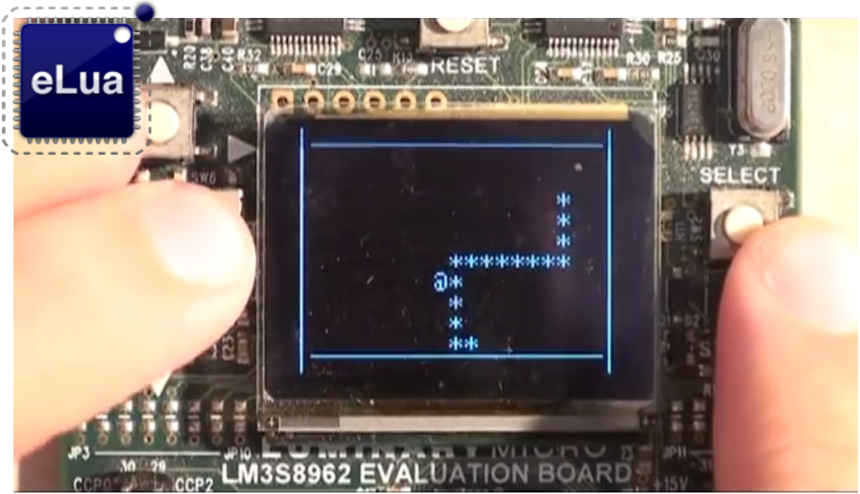
About Lua

- 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?

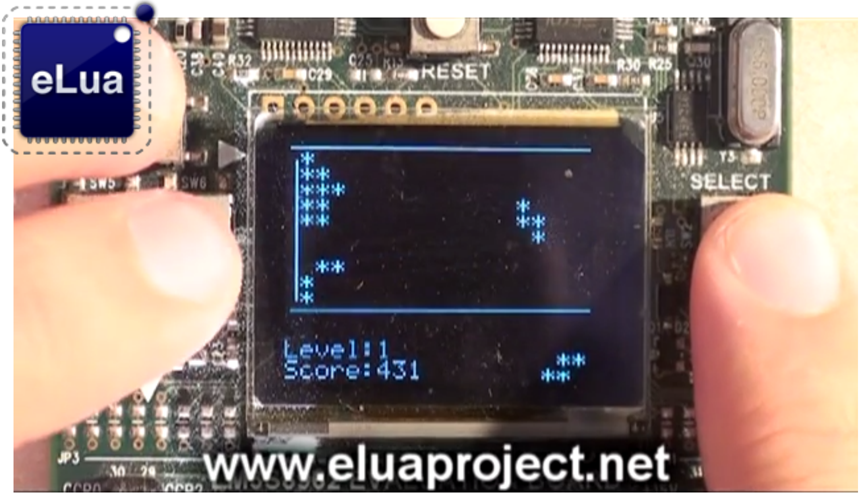
- empowers your Texas Instruments EK-LM3S to play Pong

Lua … so what?



… and Snake

Lua … so what?



… and even Tetris!

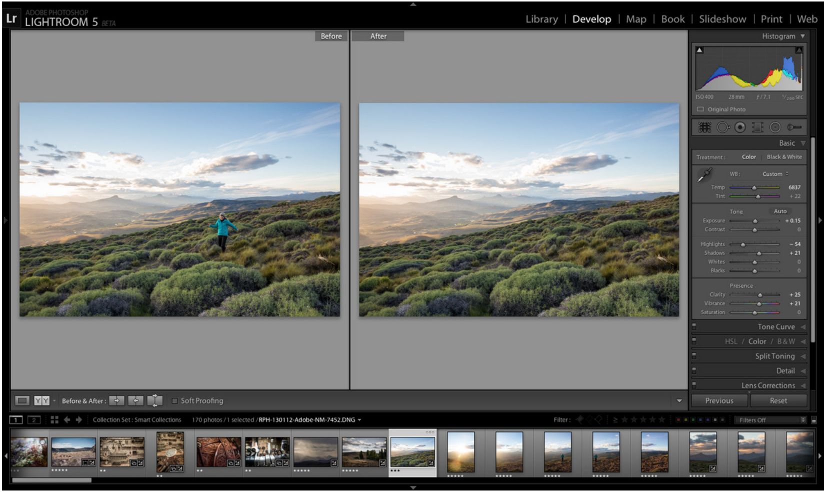
Lua … so what?

- ► 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?



- almost everywhere in Adobe's Photoshop Lightroom

Lua … so what?

awesome X11 window manager
http://awesome.naquadah.org

vim editor
http://www.vim.org

VLC Media Player
http://www.videolan.org/vlc/

LuaTeX
http://www.luatex.org

Angry Birds
http://www.angrybirds.com

Nginx HTTP Server
http://nginx.org

Wireshark
http://www.wireshark.org

NetBSD’s Kernel
http://www.netbsd.org

Havok Engine
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?

```
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 [',' '...'] | '...'
tableconstructor ::= '{' [fieldlist] '}'
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= '[' exp ']' '=' exp | Name '=' exp | exp
fieldsep ::= ',' | ';'
binop ::= '+' | '-' | '*' | '/' | '^' | '%' | '..' | '<' | '<=' | '>' | '>=' | '==' | '~=' | and | or (**)
unop ::= '-' | not | '#'
```

(**) operator precedence is missing

http://www.lua.org/manual/5.2/manual.html#9

TNG TECHNOLOGY CONSULTING

… compared to Python 3.4's Syntax

```
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER
decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef)
funcdef: 'def' NAME parameters ['->' test] ':' suite
parameters: '(' [typedargslist] ')'
typedargslist: (tfpdef ['=' test] (',' tfpdef ['=' test])* [','
  ['*' [tfpdef] (',' tfpdef ['=' test])* [',' '**' tfpdef] | '**' tfpdef]]
  | '*' [tfpdef] (',' tfpdef ['=' test])* [',' '**' tfpdef] | '**' tfpdef)
tfpdef: NAME [':' test]
varargslist: (vfpdef ['=' test] (',' vfpdef ['=' test])* [','
  ['*' [vfpdef] (',' vfpdef ['=' test])* [',' '**' vfpdef] | '**' vfpdef]]
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' '**' vfpdef] | '**' vfpdef)
vfpdef: NAME
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt | import_stmt |
  global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (augassign (yield_expr|testlist) | ('='
  (yield_expr|testlist_star_expr))*)
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
  '<<=' | '>>=' | '**=' | '//=')
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
  'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]
compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |
  classdef | decorated
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite ((except_clause ':' suite)+ ['else' ':' suite]
  ['finally' ':' suite] | 'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)* ':' suite
with_item: test ['as' expr]
except_clause: 'except' [test ['as' NAME]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
test: or_test ['if' or_test 'else' test] | lambdef
test_nocond: or_test | lambdef_nocond
lambdef: 'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ['**' factor]
atom: ('(' [yield_expr|testlist_comp] ')' |
      '[' [testlist_comp] ']' |
      '{' [dictorsetmaker] '}' |
      NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp: (test|star_expr) ( comp_for | (',' (test|star_expr))* [','] )
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [','])) |
  (test (comp_for | (',' test)* [','])) )
classdef: 'class' NAME ['(' [arglist] ')'] ':' suite
arglist: (argument ',')* (argument [','] |'*' test (',' argument)* [',' '**' test]
  |'**' test)
argument: test [comp_for] | test '=' test # Really [keyword '='] test
comp_iter: comp_for | comp_if
comp_for: 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' test_nocond [comp_iter]
encoding_decl: NAME
yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist
```

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

Basic Data Types

(1/3)

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

```
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”

```
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

(2/3)

- 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"
```

TNG TECHNOLOGY CONSULTING

For Convenience: Coercion

(3/3)

- 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()`

```
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

(1/2)

- 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

(2/2)

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

```
a = {”a”, ”b”, ”c”} -- ⇔ a = {[1]=”a”, [2]=”b”, [3]=”c”}

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

print(#a)   --> 3   -- ”length”, i.e., number of *contiguous* integer keys [💣]
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 (1/3)

- **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)
- **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

Statements and Control Structures

(2/3)

- `::<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

(3/3)

- `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”

```
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       -- declare function-local variable i = 0
   return function() -- return anonymous function with
      i = i + 1      -- variable i as upvalue
      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

Modules

(1/2)

- 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

(2/2)

- example module mod.lua

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

- usage of mod.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(coroutinefunction [,<value1>,...])` and `coroutine.yield([<value1>,...])`
- 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 (1/2)

- 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.,

```
__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 (2/2)

```
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 a metatable 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

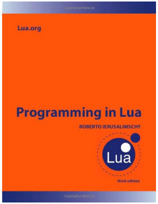
```
Prototype = {
   attribute = ”attribute value”,
   method    = function(self) print(self.attribute) end,
   new = function(self,object)
      object = object or {}            -- set or create initial object table
      self.__index = self              -- set object’s lookup to Prototype
      return setmetatable(object, self) -- return newly created object (table)
   end
}
PrototypeMT = {
   -- make Prototype table callable and invoke new() on call
   __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



~~don’t~~ try this at home!

Thank You !

? Questions

✉ christian.storm@tngtech.com