Kurs języka Lua

02 – Wciąż podstawy

Jakub Kowalski

Instytut Informatyki,
Uniwersytet Wrocławski

2017

Plan na dzisiaj

Opowiemy sobie dokładniej o

1. Kilku przydatnych konstrukcjach
2. Liczbach
3. Tablicach
4. Funkcjach

Argumenty wywołania

Tablica arg

Zmienna globalna arg, jest tablicą zawierającą informację o wywołanym skrypcie oraz podanych argumentach

```
$ lua5.3 x.lua 10 a "w w w"
arg[-1] = "lua5.3"
arg[0]  = "x.lua"
arg[1]  = "10"
arg[2]  = "a"
arg[3]  = "w w w"

$ lua -e "sin=math.sin" script a b
--> arg =
-- {[-3]="lua",    [-2]="-e", [-1]="sin=math.sin"
--    [0]="script", [1]="a",    [2]="b"}
```

Komentarze

Komentarze wielolinijkowe

- Zaczynają się od `--[[` i kończą `]]`
- (`--]]` jest tylko wielce wygodną konwencją)

```
--[[
tab[sub[2]] = 6 -- ups
--]]
```

Komentarze zagłębione i bezpieczne

- Start: `--[=`^{n}`[`, koniec: `]=`^{n}`]`, $n > 0$

```
--[===[
tab[sub[2]] = 6 -- :-)
--[[ inside commented ]]  outside commented
]===]
```

Komentarze

Komentarze wielolinijkowe

- Zaczynają się od `--[[` i kończą `]]`
- (`--]]` jest tylko wielce wygodną konwencją)

```
--[[
tab[sub[2]] = 6 -- ups
--]]
```

A co jeśli byłoby `"tab[sub[2] ]"`?

Komentarze zagłębione i bezpieczne

- Start: `--[=`^{n}`[`, koniec: `]=`^{n}`]`, $n > 0$

```
--[===[
tab[sub[2]] = 6 -- :-)
--[[ inside commented ]]  outside commented
]===]
```

Komentarze

Komentarze wielolinijkowe

- Zaczynają się od `--[[` i kończą `]]`
- (`--]]` jest tylko wielce wygodną konwencją)

```
--[[
tab[sub[2]] = 6 -- ups
--]]
```

A co jeśli byłoby `"tab[sub[2] ]"`? Działa jak zakładaliśmy!

Komentarze zagłębione i bezpieczne

- Start: `--[=`^{n}`[`, koniec: `]=`^{n}`]`, $n > 0$

```
--[===[
tab[sub[2]] = 6 -- :-)
--[[ inside commented ]]  outside commented
]===]
```

Wyrażenia boolowskie

Conditional operator

```
x = a and b or c <--> x = a ? b : c
```

Bezpieczne odwołania do zagłębionych tablic

```
zip = company and company.director and
        company.director.address and
          company.director.address.zipcode
```

Albo ładniej i efektywniej:

```
E = {} -- can be reused in similar expressions
zip = (((company or E).director or E).address
        or E).zipcode
```

Goto and swap

Goto

- Skok za pomocą konstrukcji `goto etykieta`
- W kodzie etykieta oznaczona jako `::etykieta::`

```
-- Zasymulujmy brakujące continue
while condition do
  if other_condition then goto continue end
  (some code)
  ::continue::
end
```

Swap

```
x, y = y, x
```

Lua 5.3 – Integers

Podtypy

```
print(type(2))          -- "number"
print(math.type(2))     -- "integer"
print(type(2.0))        -- "number"
print(math.type(2.0)) -- "float"
```

(`math.type` dla argumentów nie będących liczbami zwraca `nil`.)

Integer division

```
3/2       --> 1.5
3.0/2.0 --> 1.5
3//2      --> 1
3//2.0    --> 1.0
-9//2     --> -5
```

Lua 5.3 – Integers

Conversions

```
-- integer -> float
numFloat = numInteger + 0.0
-- float -> integer
numInteger = math.tointeger(numFloat)
```

tostring

```
tostring(2)    <--> '2'
tostring(2.0) <--> '2.0' -- Lua < 5.3 wypisze '2'
string.format("%d", 2.0) <--> '2'

print('11'+1) --> 12.0  (int tylko dla int args!)
```

Biblioteka math

Lua consts

```
math.huge --> inf
-- (for i=1,math.huge do ... end)
math.maxinteger -->  9223372036854775807
math.mininteger --> -9223372036854775808
```

Random

```
math.randomseed (x) -- Sets x as the RNG seed
math.randomseed (os.time()) -- time-seeding
math.random()       -- from range [0, 1)
math.random(m)      -- from range [1, m]
math.random(m, n) -- from range [m, n]
```

Biblioteka math

Rounds

```
math.floor, math.ceil
math.fmod(x, y) -- zwraca resztę z dzielenia x/y
math.modf(x) -- zwraca część całkowitą i ułamkową
```

Trygonometry functions

```
math.sin, math.cos, math.tan,
math.acos, math.asin, math.atan
```

Other standard functions

```
math.abs, math.exp, math.log, math.sqrt, math.ult
math.max, math.min, math.rad, math.deg, math.pi
```

Operator

Prefiksowy, unarny operator # zwraca długość tablicy.

Pod warunkiem, że

- Za tablicę uznamy jej fragment z kluczami naturalnymi i $\geqslant 1$
- Jeśli tablica jest niepusta to istnieje tylko jeden klucz `k` taki, że

```
t[k]~=nil and t[k+1]==nil
```

- Takie tablice nazywamy *sekwencjami*.
- Dla obliczenia długości sekwencji gwarantowana złożoność pesymistyczna jest **logarytmiczna** od jej liczby elementów.
- (algorytm oparty jest na przeszukiwaniu binarnym)

Jeśli tablica nie jest sekwencją, to (wciąż w czasie logarytmicznym) operator może zwrócić **dowolne** `k` **spełniające powyższy warunek**!

Źródła: Reference Manual, Source code

Indeksowanie

```
tab[2.0] = 10 -- float keys are converted to int
tab[2] --> 10 -- if possible
```

Klucze będące napisami

```
tab.x = 100    -- to samo co tab['x'] = 100
tab.y --> nil -- to samo co tab['y']
print (tab[x]) --> 100
```

```
tab = {}
x = 'y'
a[x] = 10
a[x] --> 10
a.x  --> nil
a.y  --> 10
```

Konstruktory (bardziej zaawansowane)

Inicjalizowanie rekordów

```
tab = {x=10, y=20}
-- is equivalent to:
tab={}; tab.x=10; tab.y=20;
```

general and mixed

```
polyline = {color='blue',     -- ['color'] = blue
            thickness=10,     -- [thickness] = 10
            ['npoints']=2,    -- npoints=2
            {x=0,  y=10},     -- polyline[1]
            {x=10, y=-1},     -- polyline[2]
            [-1]={x=0, y=0}, -- polyline[-1]
           } -- last comma is optional
```

Usuwanie

Usuwanie pól

```
tab = {1, 2, 3, 4, 5, 666}
tab[4] --> 4
tab[4] = nil
tab[4] --> nil
```

Usuwanie tablic

```
x = tab
x[6] = 36
tab = nil -- only 'x' refers to table
x[3] --> 3
x = nil -- no reference left
-- table garbage collected (eventually)
```

Iterowanie

Traversing **ALL** key-value pairs: *pairs* iterator

```
t = {10, print, x=12, k='hi', 42}
for k, v in pairs(t) do print (k, v) end
--> 1    10
--> 2    function: 68d16910
--> 3    42
--> x    12
--> k    hi
```

Traversing **ONLY** numeric fields in the sequence: *ipairs* iterator

```
for k, v in ipairs(t) do print (k, v) end
--> 1    10
--> 2    function: 68d16910
--> 3    42
```

Biblioteka table

table.insert

```
t = {11, 12, 13, 14}
table.insert(t, 15) -- t[#t+1] = 15  -- push
t --> {11, 12, 13, 14, 15}
table.insert(t, 3, 12.5) -- O(n)
t --> {11, 12, 12.5, 13, 14, 15}
```

table.remove

```
table.remove(t) -- t[#t] = nil -- pop
t --> {11, 12, 12.5, 13, 14}
table.remove(t, 3)
t --> {11, 12, 13, 14}
```

Biblioteka table

table.move

```
-- moves elements [start...end] to [newstart...]
table.move (table, start, end, newstart)
-- inserting element at the beginning
table.move(t, 1, #t, 2)
t[1] = newElement
-- removing first element
table.move(t, 2, #t, 1)
t[#t] = nil
```

table.concat

```
t = {11, 'dwanaście', 13, 14}
table.concat(t) --> 11dwanaście1314
table.concat(t,', ') --> 11, dwanaście, 13, 14
```

Działa tylko jeśli w tablicy znajdują się wyłącznie napisy lub liczby!

Prawda o tablicach

type of ...

```
print (type(math.huge))   --> number
print (type(math.sin))    --> function
print (type(math))        -->
```

Prawda o tablicach

type of ...

```
print (type(math.huge))   --> number
print (type(math.sin))    --> function
print (type(math))        --> table
print (type(table))       --> table
print (type(io))          --> table
```

W Lua tablice są podstawą do prawie wszystkich rozwiązań...
W szczególności moduły, to po prostu tablice z odpowiednio nazwanymi polami.
Dlatego w dowolnym miejscu programu można zrobić np. tak:

```
local m = math
print (m.sin(1)) --> 0.8414709848079
math.pi = 4;  math.beast = 666
print (math.pi, math.beast) --> 4  666
```

Call shortcuts

Jeśli funkcja bierze jeden argument, który jest albo napisem albo konstruktorem tablicy, to nawiasy są opcjonalne.

```
print 'Hello World'    <--> print('Hello World')
print [[ a multi-line <--> print([[a multi-line
 message]]                     message]])
f{x=10, y=2}          <--> print({x=10, y=2})
type{}                <--> type({})
```

'Named' arguments

Funkcje które biorą 'nazwane' argumenty, możemy sumulować za pomocą funkcji których argumentem jest tablica.

```
function rename(args)
  return os.rename(args.old, args.new)
end
rename{ new = "perm.lua", old = "temp.lua" }
```

Multiple arguments

Arguments behavior

```
function f (a, b) print(a, b) end
```

Multiple arguments

Arguments behavior

```
function f (a, b) print(a, b) end
f()       --> nil   nil
f(3)      --> 3     nil
f(3,4)    --> 3     4
f(3,4,5)  --> 3     4     -- 5 is discarded
```

Default arguments

```
function tabinsert (tab, index)
  -- default value if argument is nil:
  local index = index or #tab+1
  ...
end
```

Multiple results

```
function foo00 () end
function foo21 (a,b) return a+b end
function fooT2 (tab)
  return tab[1], tab[#tab] end -- 2 results
```

Rules

- Function called as statement:
  discards all results
- Function called in expression (e.g. +):
  only the first result keeped
- Called as last/only expression in
  - multiple assignemnts,
  - arguments to function calls,
  - table constructors,
  - return statements:

  get all the results

Multiple results

Multiple assignments

```
t = {3, 4, 5}
x, y = fooT2(t)          --> x=3,   y=5
x = fooT2(t)             --> x=3,   5 is discarded
x, y = foo00()           --> x=nil, y=nil
x, y = foo21(1,1)        --> x=2,   y=nil

x, y, z = 1, fooT2(t)      --> x=1, y=3, z=5
x, y, z = fooT2(t), 6, 9 --> x=3, y=6, z=9
x, y = fooT2(t), 6, 9      --> x=3, y=6
```

Expressions

```
3 + fooT2(t)    --> 6
fooT2(t) .. 'a'   --> 3a
```

Multiple results

Function call arguments

```
foo21(1, 2, 3, 4)    --> 3
foo21(fooT2(t))      --> 8
foo21(fooT2(t), 10) --> 13
```

Table constructors

```
t2 = {foo00()}                  --> t2 = {}
t2 = {1, fooT2(t)}              --> t2 = {1, 3, 5}
t2 = {fooT2(t), foo00(), 7} --> t2 = {3, nil, 7}
```

Multiple results

Return statements

```
function fooT3(t) return #t, fooT2(t) end
fooT3(t)    --> 3, 3, 5

local function range(a, b, c)
  if a > b then return
  else return a, range(a + c, b, c) end
end
print(range(1, 9, 2))   --> 1 3 5 7 9
```

Forcing one result return

```
print( ( foo00() ) )    --> nil
print( ( fooT2(t) ) )   --> 3     -- additional
print( ( fooT3(t) ) )   --> 0     -- parentheses!
```

Variadic functions

... expression

```
function add (a, ...)
  local s = a or 0
  for _, v in ipairs{...} do s = s + v end
  return s
end
add(3, 4, 10, 25, 12) --> 54

function foo (a, b, c)
-- is equal to
function foo (...)
  local a, b, c = ...
```

Na głównym poziomie wyrażenie ... zawiera argumenty (arg[>0]).

Packing, Unpacking, Selecting

table.pack

```
table.pack(4, 5, 6) --> {4, 5, 6, n=3}
table.pack(4, nil, 6) --> {[1]=4, [3]=6, n=3}
table.pack(nil, nil)  --> {n=2}
```

(Pole `.n` jest konwencją na 'prawdziwą' wielkość tablicy)

select

Zwraca wszystkie argumenty od *n*-tego (ich liczbę jeśli #).

```
select(1, 'a', 'b', 'c')    --> return a, b, c
select(2, 'a', 'b', 'c')    --> return b, c
select(-2, 'a', 'b', 'c')   --> return b, c
select('#', 'a', 'b', 'c') --> return 3
```

Packing, Unpacking, Selecting

table.unpack

Odwrotność pack – tworzy listę parametrów z tablicy

```
a, b = table.unpack{10, 20, 30} --> a=10, b=20
print(table.unpack{4, 5, 6, 7}) --> 4  5  6  7
-- additional arguments min=1, and max=#table
table.unpack({4, 5, 6, 7, 8}, 2, 4)  --> 5, 6, 7
```

Funkcja ta pozwala na *generic calls*, dynamiczne wywoływanie dowolnej funkcji z dowolnymi parametrami.

```
function unpack (t, i, n)
  local i = i or 1
  local n = n or #t
  if i<=n then
    return t[i], unpack(t, i+1, n)
  end
end
```

Tail calls

tail-call elimination

Wywołania funkcji które interpreter obsługuje przez goto.
Dzięki temu takie wywołanie nie zajmuje miejsca na stosie.

Proper tail calls

```
function f (x) x=x+1; return g(x) end
function foo(n) if n>0 then return foo(n-1) end end
return x[i].foo(x[j]+a*b*666, i+127*j)
```

Not proper tail calls

```
function f(x) g(x) end --must discard g's results
return g(x) + 1    -- must do addition
return x or g(x)   -- must adjust to one result
return (g(x))      -- must adjust to one result
```

Dziękuję za uwagę

Za tydzień:
napisy,
input/output,
pattern-matching,
moduły,
data i czas
programowanie funkcyjne.

Intuicje dotyczące IO

io.read – type

```
io.read('*number')        --
```

Intuicje dotyczące IO

io.read – type

```
io.read('*number')        --  number
```

Intuicje dotyczące IO

io.read – type

```
io.read('*number')          --  number
io.read('*nil')             --
```

Intuicje dotyczące IO

io.read – type

```
io.read('*number')          --  number
io.read('*nil')             --  number
```

Intuicje dotyczące IO

io.read – type

```
io.read('*number')        --  number
io.read('*nil')           --  number
io.read('*no pasaran')    --
```

Intuicje dotyczące IO

io.read – type

```
io.read('*number')        --  number
io.read('*nil')           --  number
io.read('*no pasaran')    --  number
```

Intuicje dotyczące IO

io.read – type

```
io.read('*number')        --  number
io.read('*nil')           --  number
io.read('*no pasaran')    --  number
io.read('na na na na na na na na na na BATMAN!')
                          --
```

Intuicje dotyczące IO

io.read – type

```
io.read('*number')        --  number
io.read('*nil')           --  number
io.read('*no pasaran')    --  number
io.read('na na na na na na na na na na BATMAN!')
                          --  number   :-)
```