Kurs języka Lua

05 – Funkcje

Jakub Kowalski

Instytut Informatyki,
Uniwersytet Wrocławski

2017

Plan na dzisiaj

Opowiemy sobie dokładniej o

1. Domknięciach
2. Iteratorach
3. Struktury danych

Domknięcia

Funkcje

Są wartościami pierwszego rzędu

Czyli zachowują się jak wszystkie inne wartości – przechowujemy w zmiennych lokalnych/globalnych, zwracamy funkcjach, dajemy jako argumenty funkcji, wstawiamy do tablic itd.

Korzystają z leksykalnego zasięgu

Czyli zasięg zmiennej jest zdefiniowany przez jej położenie w kodzie, a zagnieżdżone funkcje mają dostęp do zmiennych zadeklarowanych w ich obszarze zewnętrznym.

- Funkcje są zawsze przekazywane przez referencję
- Wszystkie funkcje są anonimowe
  (istnieją tylko przez przypisanie do nazwanej zmiennej)

```
foo = function (x) return 2*x end
```

Sortowanie

table.sort (list [, comp])

- Sortuje zadaną listę w miejscu
- Domyślnie funkcją porównującą jest <
- Funkcja porównująca powinna zwracać true jeśli pierwszy argument a być przed drugim

```
tuples = {
  {x = 1, y = 2}, {x = 0.5, y = 2.8},
  {x = 1, y = -4}, {x = -2, y = 2}
}

table.sort(tuples, function (a,b)
    return a.x>b.x or (a.x==b.x and a.y>b.y) end)
-->   1     2
-->   1    -4
-->   0.5   2.8
--> -2     2
```

Funkcje wyższych rzędów

$$f'(x) = (f(x+d) - f(x))/d$$

```
function derivative (f, delta)
  delta = delta or 1e-4
  return function (x)
           return (f(x+delta) - f(x))/delta
         end
  end

c = derivative (math.sin)

print (math.cos(5.2), c(5.2))
--> 0.46851667130038    0.46856084325086
print (math.cos(10), c(10))
--> -0.83907152907645   -0.83904432662041
```

Funkcje nieglobalne

Deklarowanie funkcji w tablicy

```
Lib = {}
Lib.foo = function(x,y) return x + y end
Lib = {foo = function(x,y) return x + y end}
function Lib.foo (x,y) return x + y end
```

Rekursja w lokalnej funkcji

```
local fact = function (n)
  if n==0 then return 1
  else return n*fact(n-1) end
end
-- attempt to call a nil value (global 'fact')
```

Funkcje nieglobalne

Rekursja w lokalnej funkcji – rozwiązanie

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

Specjalna składnia

```
local function foo(<params>) <body> end
<-->
local foo; foo = function (<params>) <body> end
```

Funkcje nieglobalne

Nawzajemne wywołania

```
local f -- deklaracja poprzedzająca

local function g()
   <some code>   f()   <some code>
end

function f()
   <some code>   g()   <some code>
end
```

Dlaczego "`function f()`" nie ma `local`?

Funkcje nieglobalne

Nawzajemne wywołania

```
local f -- deklaracja poprzedzająca

local function g()
   <some code>   f()   <some code>
end

function f()
   <some code>   g()   <some code>
end
```

Dlaczego "`function f()`" nie ma `local`?

```
attempt to call a nil value (upvalue 'f')
```

Zasięg leksykalny

Upvalues

```
names = {'Adam', 'Ala', 'Abel'}
grades = {Abel = 8, Adam = 5, Ala = 10}
table.sort(names, function (n1, n2)
  return grades[n1] > grades[n2] end)
end
```

Zasięg leksykalny

Upvalues

```
names = {'Adam', 'Ala', 'Abel'}
grades = {Abel = 8, Adam = 5, Ala = 10}
table.sort(names, function (n1, n2)
  return grades[n1] > grades[n2] end)
end
function sortbygrade (names, grades)
  table.sort(names, function (n1, n2)
    return grades[n1] > grades[n2] end)
end
```

- Anonimowa funkcja w `sort` ma dostęp do zmiennej `grades`, która jest parametrem okalającej ją funkcji `sortbygrade`
- Wewnątrz tej funkcji anonimowej, `grades` nie jest ani zmienną lokalną ani globalną
- Jest zmienną nielokalną (*non-local variable*)
- Takie zmienne nazywane są *upvalues*

Domknięcia

Dzięki funkcjom, będącym wartościami pierwszego rzędu, zmienne mogą uciekać swojemu oryginalnemu zasięgowi.

```
function newCounter ()
  local i = 0
  return function ()
           i = i + 1
           return i
         end
end
c1 = newCounter()
print(c1()) --> 1
print(c1()) --> 2
```

Domknięcia

Dzięki funkcjom, będącym wartościami pierwszego rzędu, zmienne mogą uciekać swojemu oryginalnemu zasięgowi.

```
function newCounter ()
  local i = 0
  return function ()
           i = i + 1
           return i
         end
end
c1 = newCounter()
print(c1()) --> 1
print(c1()) --> 2
c2 = newCounter()
print(c2()) -->
```

Domknięcia

Dzięki funkcjom, będącym wartościami pierwszego rzędu, zmienne mogą uciekać swojemu oryginalnemu zasięgowi.

```
function newCounter ()
  local i = 0
  return function ()
           i = i + 1
           return i
         end
end
c1 = newCounter()
print(c1()) --> 1
print(c1()) --> 2
c2 = newCounter()
print(c2()) --> 1    -- ta sama funkcja
print(c1()) --> 3
print(c2()) --> 2    -- ale inne domknięcie
```

Domknięcia

Closure

Wartością w Lua jest nie funkcja ale domknięcie – czyli funkcja plus wszystko czego funkcja potrzebuje żeby mieć poprawny dostęp do swoich nielokalnych zmiennych

callbacks

Funkcja wywoływana w momencie podjęcia akcji przez użytkownika.

```
function DigitButton (digit)
  return Button{ label = digit,
                 action = function ()
                   add_to_display(digit)
                 end
               }
end
```

Redefiniowanie funkcji

Dzięki domknięciom możemy wygodnie redefiniować funkcje, wciąż korzystając z ich oryginalnych implementacji.

```
-- Chcemy żeby math.sin operowała na stopniach
do
  local old_sin = math.sin
  local k = math.pi/180
  math.sin = function (x)
               return old_sin(x*k)
             end
end
```

Dzieki zamknięciu kodu w blok `do ... end` ograniczamy zasięg zmiennej `oldSin`, a więc dostęp do niej jest od teraz możliwy tylko poprzez naszą nową funkcję `sin`.

Redefiniowanie funkcji

Tworzenie bezpiecznych środowisk

```
do
  local oldOpen = io.open
  local access_OK = function (filename, mode)
                      <check access>
                    end
  io.open = function (filename, mode)
      if access_OK(filename, mode) then
        return old_open(filename, mode)
      else
        return nil, "access denied"
      end
end
```

- Nie ma już niezabezpieczonego dostępu do oryginalnego `io.open`
- Więc mamy prostą i wygodną metodę sandboxingu

Podejście funkcjonalne

Prosty system dla regionów geometrycznych

```
function disk (cx, cy, r)
  return function (x, y)
           return (x-cx)^2+(y-cy)^2<=r^2
         end
end

function rect (left, right, bottom, up)
  return function (x, y)
           return left <= x and x <= right and
                  bottom <= y and y <= up
         end
end
```

Podejście funkcjonalne

```
function complement (r)
  return function (x, y)
           return not r(x, y)
         end
end

function translate (r, dx, dy)
  return function (x, y)
           return r(x-dx, y-dy)
         end
end

function difference (r1, r2)
  return function (x, y)
           return r1(x, y) and not r2(x,y)
         end
end
```

Podejście funkcjonalne

Księżyc przybywający

```
c1 = disk(0, 0, 1)
plot(
     difference(c1, translate(c1, 0.3, 0)),
     500, 500
    )
```

Iteratory

Iteratory

- Iterator to dowolna konstrukcja pozwalająca na przechodzenie po elementach kolekcji.
- W Lua iteratory są zazwyczaj funkcjami, których każdorazowe wywołanie zwraca „następny” element kolekcji.
- Znane nam przykłady:

  `pairs`, `ipairs`, `io.read`, `io.lines`, `string.gmatch`

- Iteratory muszą trzymać swój stan pomiędzy kolejnymi uruchomieniami, np. korzystając z domknięć.
- Takie domknięcie zazwyczaj składa się z dwóch funkcji: samego domknięcia oraz *factory*, czyli funkcji tworzącej domknięcie wraz z zawartymi w nim zmiennymi

Prosty iterator

values za każdym wywołaniem tworzy nowy closure

```
function values (t) -- factory
  local i = 0
  return function () i = i + 1; return t[i] end
end

t = {10, 20, 30}
iter = values(t) -- creates the iterator
while true do
  local e = iter() -- calls the iterator
  if e == nil then break end
  print (e)
end
<-->
for e in values(t) do
  print (e)
end
```

Bardziej skomplikowany iterator

```
function allwords ()
  local line = io.read() -- current line
  local pos = 1          -- position in line
  return function ()     -- iterator function
    while line do
      local w, e = line:match('(%w+)()', pos)
      if w then          -- found a word?
        pos = e          -- set next position
        return w         -- return the word
      else
        line = io.read() -- try next line
        pos = 1          -- reset position
      end
    end
    return nil           -- no more lines - end
  end
end
```

Ale wciąż proste wywołanie

```
for word in allwords() do
  print ('#'..word..'#')
end
```

Generyczny for

```
for <var-list> in <exp-list> do
  <body>
end
```

- *var-list* jest listą oddzielonych przecinkami nazw zmiennych
- pierwsza zmienna jest *zmienną kontrolną*, kiedy osiągnie wartość `nil` pętla się kończy
- *exp-list* jest listą oddzielonych przecinkami wyrażeń
- najpierw jest ewaluowane *exp-list* które powinno zwrócić 3 wartości

Generyczny for

Semantyka

```
for var_1, ..., var_n in <explist> do <block> end
<-->
do
  local _f, _s, _var = <explist>
  while true do
    local var_1, ..., var_n = _f(_s, _var)
    _var = var_1
    if _var == nil then break end
    <block>
  end
end
```

- funkcja iterująca: f
- stan niezmienniczy: s
- zmienna kontrolna: a_0
- kolejne wywołania: $a_1 = f(s, a_0)$; $a_2 = f(s, a_1)$; ...; $nil = f(s, a_n)$

Przykładowy iterator (closure)

fromto

```
function fromto(a, b)
  return function ()
           if a > b then return nil
           else a = a + 1; return a - 1 end
         end
end
for i in fromto(2, 5) do print (i) end
--> 2\n3\n4\n5

print (fromto(2, 5)) --> function: 0038f198
print (fromto(3, 7)) --> function:
```

Przykładowy iterator (closure)

fromto

```
function fromto(a, b)
  return function ()
           if a > b then return nil
           else a = a + 1; return a - 1 end
         end
end
for i in fromto(2, 5) do print (i) end
--> 2\n3\n4\n5

print (fromto(2, 5)) --> function: 0038f198
print (fromto(3, 7)) --> function: 0038f1f8
```

- najprostszym sposobem na zdefiniowanie iteratora jest użycie domknięć
- a, b są stanem iteratora
- iteratorem jest **domknięcie** zwracane przez fromto

Iteratory bezstanowe

ipairs

```
local function iter (t, i) -- iter istnieje
  i = i + 1
  local v = t[i]
  if v then
    return i, v
  end
end
function ipairs (t)
  return iter, t, 0
end

print (ipairs(a))
--> function: 0038cc08  table: 0038b190 0
print (ipairs(t))
--> function: 0038cc08  table: 0038b168 0
-- stateless iterator, external state, seed
```

Iteratory bezstanowe

pairs

```
print (next) --> function: 68d16a80

function pairs (t)
  return next, t, nil
end

for k, v in pairs(a) do
  print (k, v)
end
<-->
for k, v in next, a do
  print (k, v)
end
```

Trawersowanie listy

```
local function getnext (node)
  return node.next
end
function traverse (list)
  return getnext, nil, list
end
```

Trawersowanie listy

```
local function getnext (node)
  return node.next
end
function traverse (list)
  return getnext, nil, list
end  -- pominiemy pierwszy element!
```

```
local function getnext (list, node)
  if not node then return list
  else return node.next end
end
function traverse (list)
  return getnext, list, nil
end -- pierwszy node jako 'invariant state'
```

Przykładowy iterator (stateless)

fromto

```
function fromto(a, b)
  return function (state, seed)
           if seed >= state then
             return nil
           else
             return seed + 1
           end
         end, b, a-1
end

print(fromto(2, 5)) --> function: 0035cc80   5   1
print(fromto(3, 7)) --> function: 0035cc80   7   2
```

- Nie tworzymy „prawdziwego” domknięcia

Przykładowy iterator (seedless)

```
function fromto(a, b)
  return function (state)
           if state[1] > state[2] then
             return nil
           else
             state[1] = state[1] + 1
             return state[1] - 1
           end
         end, { a, b }
end
print (fromto(2, 5))
--> function: 00afccb0   table: 00afb5a0
print (fromto(3, 7))
--> function: 00afccb0   table: 00afb668
```

- używamy mutowalnych wartości jako stanu
- możemy wtedy wykorzystać stan do trzymania seeda

Key-ordered table traversal

```
function pairsByKeys (t, f)
  local a = {}
  for n in pairs(t) do
    a[#a + 1] = n
  end
  table.sort(a, f)
  local i = 0
  return function ()
    i = i + 1
    return a[i], t[a[i]] -- return key, value
  end
end

for k, v in pairsByKeys({a=10, b=5, c=20}) do
  print (k, v)
end
```

Prawdziwe iteratory

- Dotychczas pokazane iteratory nie iterują :(
- Całą robotę odwala za nie `for`... czas to zmienić!
- Zrobmy tak żeby iterator przyjmował funkcję która wie co robić

```
function allwords (f)
  for line in io.lines() do
    for word in line:gmatch('(%w+)') do
      f(word)    -- wywołanie funkcji
    end
  end
end

local count = 0
allwords (function (w)
            if w=='hello' then count=count+1 end
          end)
print (count)
```

Różnice

- „prawdziwe iteratory” były popularne w starszych wersjach Lua kiedy nie było pętli `for`
- oba style mają podobny narzut złożonościowy
- pisanie prawdziwych iteratorów wydaje się prostsze jeśli chodzi o kod
- „generatory” natomiast wygodniej jest zrównoleglić
- oraz dają możliwość korzystania z `break` i `return`

Struktury danych

Struktury danych w Lua

Wcześniej to ukrywałem, ale w Lua jest cała masa struktur danych...

- tablice
- macierze
- listy
- kolejki
- zbiory
- bufory napisów
- grafy

Struktury danych w Lua

Wcześniej to ukrywałem, ale w Lua jest cała masa struktur danych...

- tablice

No dobra, żartowałem.

Dziękuję za uwagę

Za tydzień:
moduły,
metatabele,
wstęp do obiektowości,
...?