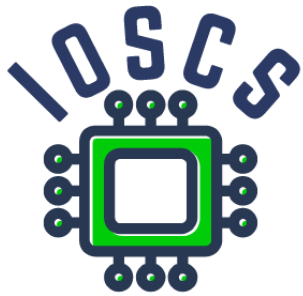


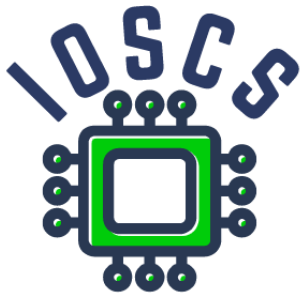
Project: Innovative Open Source Courses for Computer Science

Programovací jazyk Lua
Učebný materiál

Tomáš Hála
Mendel University in Brno

29. 05. 2021

Innovative Open Source Courses for Computer Science



This teaching material was written as one of the outputs of the project "Innovative Open Source Courses for Computer Science", funded by the Erasmus+ grant no. 2019-1-PL01-KA203-065564. The project is coordinated by West Pomeranian University of Technology in Szczecin (Poland) and is implemented in partnership with Mendel University in Brno (Czech Republic) and University of Žilina (Slovak Republic). The project implementation timeline is September 2019 to December 2022.

Project information

Project was implemented under the Erasmus+.
Project name: **"Innovative Open Source courses for Computer Science curriculum"**
Project nr: **2019-1-PL01-KA203-065564**
Key Action: **KA2 – Cooperation for innovation and the exchange of good practices**
Action Type: **KA203 – Strategic Partnerships for higher education**

Consortium
ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY W SZCZECINIE
MENDELOVA UNIVERZITA V BRNE
ZILINSKA UNIVERZITA V ZILINE

Co-funded by the
Erasmus+ Programme
of the European Union

Programovací jazyk Lua

Tomáš Hála

Učebný materiál

Funded by
the European Union

úvod

algoritmy – vlastnosti

- jednoznačný (deterministický)
- konečný, t. j. vždy vedie k určitým výsledkom
- obecný, t. j. použiteľné na riešenie daného problému pomocou akýchkoľvek prípustných údajov
- opakovatelný, t. j. vedie vždy k rovnakým výsledkom s rovnakými vstupnými údajmi

algoritmy – vyjadrenie

- verbálne – v prirodzenom jazyku
- graficky – vývojový diagram alebo štruktúrogram
- matematicky – vzťah medzi veličinami, sústava rovníc, matíc
- programovací jazyk

algoritmizácia

- vstupom je problem
- výstupom je algoritmus

programovanie

- vyjadrenie algoritmu
- programovací jazyk(y)
- ladenie
- testovanie
- vstupné dáta, výstupné informácie

programovacie jazyky

- program v programovacom jazyku: čitateľný človekom ale počítač tomu nerozumie
- strojový kód
- kompilácia, kompilátor
- interpretované programy, tlmočníci

o jazyke Lua

história

- 1993
- Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes,

  Computer Graphics Technology Group (Tecgraf), Pontifical Catholic University of Rio de Janeiro, Brazília
- multiparadigmatický:
  - scripting
  - imperatívny (procedurálny, prototypovací, objektovo orientovaný)
  - funkcionálny

verzie

- 5.1.x
- 5.2.x
- 5.3.x (tu použitá)
- 5.4.3 (posledná existujúca)

zdroje

```
 apt install lua
...
Package lua is a virtual package provided by:
  lua5.3:i386 5.3.3-1ubuntu0.18.04.1
  lua5.3 5.3.3-1ubuntu0.18.04.1
  lua5.2:i386 5.2.4-1.1build1
  lua5.1:i386 5.1.5-8.1build2
  lua50 5.0.3-8
  lua5.2 5.2.4-1.1build1
  lua5.1 5.1.5-8.1build2
You should explicitly select one to install.
```

online

https://geekflare.com/online-compiler/lua

https://www.jdoodle.com/execute-lua-online/

https://onecompiler.com/lua/3y5j9aajb

https://replit.com/languages/lua

https://www.tutorialspoint.com/execute_lua_online.php

https://www.lua.org/demo.html

štruktúra jazyka

štruktúra – 22 vyhradených slov

```
and       break     do        else
elseif    end       false     for
function  goto      if        in
local     nil       not       or
repeat    return    then      true
until     while
```

konštanty	`false true nil`
premenné	`local`
operátory	`and not or`
vetvenie	`if then else elseif end`
cykly	`for in repeat until while do end`
funkcie	`function return`
skoky	`break goto`

vetvenie – if

```
if condition then ... end
if condition then ... else ... end

if     condition1 then ...
elseif condition2 then ...
elseif condition3 then ...
else ...
end
```

cykly – while

```
while condition do
  ...
end
```

Príklad:

```
i=0
while i<10 do
  i=i+1
  print (i, i*2, i^3)
end
```

cykly – repeat

```
repeat
  ...
until condition

x,i = 1,5
repeat
   x=x*i
   i=i-1
until i==1
print (x)
```

cykly – for I.

```
for variable=start,stop do
 ...
end

for i=1,10 do
  print (i, i*2, i^3)
end
```

cykly – for II.

```
for variable=start,stop,step do
  ...
end

for i=-10,10,2 do
  print (i, i*2, i^3)
end
```

dátové typy a operátory

všeobecne

- každá premenná v programe je určitého typu
- každý programovací jazyk definuje vlastnú množinu dátových typov

dátový typ tvorí definície:

- množiny hodnôt
- vnútornej reprezentácie hodnôt v počítači (veľkosť v pamäti, kódovanie hodnôt)
- množiny prípustných hodnôt

porovnaj:

- Napr. (Pascal) `var x:boolean;`
  - hodnoty: true, false
  - pamäť: 1 byte, bit 0 je rozhodujúci
  - operácie: not, and, or

v jazyku Lua

- dynamicky typovaný jazyk
- nie sú definície typov
- hodnoty nesú informáciu o príslušnosti k dátovému typu
- osem základných dátových typov:
  - nil
  - boolean, number, string
  - function
  - table
  - userdata, thread (tu nepreberáme)

nil

- nil je nil :-)
- líši sa od všetkých ostatných hodnôt
- značí neprítomnosť užitočnej hodnoty

boolean

- false a true

number

- pre celočíselné hodnoty i hodnoty s pohyblivou rádovou čiarkou
- 8 B
- najväčšia hodnota?

string

- immutabilná postupnosť bytov
- reťazec môže obsahovať akúkoľvek osembitovú hodnotu, vrátane '\0'
- nepredpokladá sa žiadné kódovanie

number – aritmetické operácie

+	sčítanie
–	odčítanie
*	násobenie
/	delenie
//	celočíselné delenie
%	modulo (zvyšok po delení)
^	umocňovanie
–	unárne mínus

number – bitové operácie

~	unárne bitové NOT
&	bitové AND
|	bitové OR
~	bitové exkluzívne OR
>>	bitový posun doprava
<<	bitový posun doľava

boolean – logické operácie

- not
- and or

- obvyklý význam
- používane tiež pri ostatných typoch

string – spojenie, dĺžka

- ..
- čísla sa konvertujú na string
- #
- number of *bytes*

- # pozri tabuľky

relačné operátory

==	rovnosť
~=	nerovnosť
<	menšie ako
<=	menšie alebo rovné
>	väčšie ako
>=	väčšie alebo rovné

priorita operátorov (od najvyššej po najnižšiu)

```
^

unárne operátory (not   #     -     ~)

*     /     //    %

+     -

..

<<    >>

&

~

|

<     >     <=    >=    ~=    ==

and

or
```

funkcie

terminológia

- funkcie v procedúry
- deklarácia (hlavička funkcie)
- parametre (nedá sa použiť na úpravu hodnôt v hlavnom bloku)
- návratové hodnoty

terminológia

- preddefinované vs. vlastné funkcie
- body: odporúčané sú lokálne premenné
- špecifický druh: iteračné funkcie (vysvetlené budú neskôr)

funkcie – príklad

```
function perimeter_triangle (a, b, c)
  return a+b+c
end
```

S návratovou hodnotou

```
function myprint (a, b, c)
  print("value a is", a)
  print("value b is", b)
  print("value c is", c)
end
```

Bez návratovej hodnoty

rekurzívne funkcie

- funkcia volá sama sebe
- každá definícia rekurzívneho algoritmu musí obsahovať hodnotu pre ukončenie rekurzie (hodnota 1 v nasledujúcom príklade)
- efektívne?

rekurzívne funkcie

- priama rekurzia: volá sa priamo
- nepriama rekurzia: sú potrebné dve alebo viac funkcií –
  F1 volá F2 a F2 volá F1

rekurzívne funkcie – príklad

```
function GCD(x, y)
  if     x==y then return x
  elseif x>y then return GCD(x-y,y)
  else return GCD(x,y-x)
  end
end

-- v programe:

cislo1, cislo2 = io.read("*n", "*n")
print(GCD(cislo1, cislo2))
```

reťazce znakou

nástroje

reťazec je objekt
metódy:

```
.. -- zreťazenie
string.len(arg) -- dĺžka
string.rep(s, n)) -- replikuje reťazec n-krát

string.upper(s)
string.lower(s)

string.reverse(s)
string.char(x) -- obj. hodnota --> znak/reťazec
string.byte(ch) -- char --> ord. hodnotu
```

formátovanie výstupu

```
string.format(...)
```

nástroje: hľadať & nahradiť

```
string.gsub( s , fs , rs)
```

Vráti reťazec nahradením výskytov `fs` za `rs`.

```
string.gmatch( s, vzor)
```

Vráti fragmenty `s` popísané pomocou `pattern`.

```
string.find ( s , fs [, startindex , endindex] )
```

Vráti počiatočný index a koncový index `fs` v `s` (alebo `nil`, ak sa nenájde).

nástroje: hľadať & nahradiť

```
string.sub ( s , startindex , endindex )
```

startindex je i-tý index endindex je j-tý index posledného indexu reťazca, ktorý chceme

```
s = "This is my text."
print(string.sub(s, 2, 3))
print(string.sub(s, 2, -2))
```

chýbajúce nástroje

- `split` napríkald na rozdelenie CSV dát
- `trim` na odstranenie koncových medzier

- môžeme písať vlastné funkcie

Lua vzory

Regulárny výraz POSIX vs. vzory Lua

Lua vzory

triedy:

```
. všetky znaky
%a písmena
%c riadiacie znaky
%d číslice
%l malá písmená
%p interpunkčná znamienka
%s medzery
%u veľká písmená
%w alfanumerické znaky
%x hexadecimálne číslice
%z znak s ordinální hodnotou 0
```

Lua patterns

dopĺňky do množín:

```
%A písmen
%C riadiacich znakov
%D číslic
%L malých písmen
%P interpunkčných znakov
%S medzer
%U veľkých písmen
%A alfanumerických znakov
%X hexadecimálnych číslic
%Z znaku s ordinální hodnotou 0
```

Lua patterns

escape, kotvy, iterátory (modifikátory), množiny + skupiny:

```
%
^ $
+ - * ?
[ ] ( ).
```

split

kód funkcie mysplit

```
function mysplit_print( s , sep )
  for i in s:gmatch("([^"..sep.."]*)") do
    print (i)
  end
end

function mysplit( s , sep )
  local t = {}
  for i in s:gmatch("([^"..sep.."]*)") do
    table.insert(t,i)
  end
  return t
end  -- zjednodušené verzie len pre neprázdne polia
```

rozdeliť

verzia so zachovánim prázdných polí:

```
function split ( s, sep )
  sep = sep or '%s'
  local t = {}
  for field,s in string.gmatch
        ( s, "([^"..sep.."]*)("..sep.."?)") do
    table.insert(t, field)
    if s=="" then return t end
  end
end
```

rozdeliť – použiť

```
a = "John:Smith:1999:10:21:London:UK"
mysplit_print(a,":")
t = mysplit (a,":")
for i=1,#t do print(t[i]) end
```

rozdeliť – použiť

špinavý trik:

```
string.split = mysplit
t = a:split(":")
```

[možné, ale neodporúča sa]

štruktúrované datové typy

obecne

- (indexované) pole
- záznam / struct
- bitové pole (set)
- asociatívne pole (hash)

- objekt

- A čo v Lua?

table

konštruktory

```
t={}

t[1]=1
t[2]=2
t[3]=7

t={1,2,7,5,13,-1}
t={1,2,7,5,13,-1,}
```

= homogénne pole, indexované

konštruktory II.

```
t={}
t={1,2,7,"Lua",true,{},-9.9999,false,8888}
```

= heterogénne pole, indexované

konštruktory III.

```
t={}

t["jan"]=31
t["feb"]=28
t["mar"]=31

t.jan=31
t.feb=28
t.mar=31
```

= asociatívne pole (hash)

konštruktory IV.

```
m="jan"
t[m]=31 -- vs. t.m (!)
...

t={jan=31,feb=28,mar=31}
```

... kombinácia indexovaného a asociatívneho poľa

knižnica table

- table.insert
- table.remove
- table.sort
- #

knižnica table II.

```
t={}
table.insert(t,"Monday")
table.insert(t,"Tuesday")
table.insert(t,"Wednesday")

for i=1,#t do
   print(t[i])
end
```

tabuľka – výstup

```
for k,v in pairs(t) do
  print(k,v)
end
```

prevod poľa na hash

```
a = { 1, 2, 3, 4 }

h = {}
for i=1,#a do h[a[i]]=true end
```

výsledkom je množina

prevod poľa na hash

```
a = { 1, 2, 3, 4, 1, 3, 3, 4 }

h = {}
for i=1,#a do h[a[i]]=(h[a[i]] or 0) + 1 end
```

výsledkom je multimnožina

vzostupne, zostupne, alebo...?

```
a = { "January", "February", "March", "April",
      "June",    "July",     "August"
    }

table.sort(a)
table.sort(a, function (x,y) return y<x end)

table.sort(a,
        function (x,y) return x:len()<y:len() end
        )
table.sort(a,
        function (x,y) return x:reverse()<y:reverse() end
        )
```

funkcie II.

funkcia ako dátový typ

```
function f1 (a,b)   return a+b   end
function f2 (a,b)   return a-b   end
f=f1   print(f(3,5))
f=f2   print(f(3,5))

function domath(a,b,f)
  return f(a,b)
end

print (domath(4,7,f1))
print (domath(4,7,f2))
```

funkcia ako dátový typ

```
a = { "January", "February", "March", "April",
      "June",    "July",     "August"
    }

table.sort(a,
        function (x,y) return x:reverse()<y:reverse() end
        )

function mysort(x,y)
   return x:reverse()<y:reverse()
end

table.sort(a, mysort)
```

funkcie: iterátory a uzávery

- iteračná funkcia umožňuje prechádzanie cez dáta (tabuľky, súbory)
- dve funkcie:
  - iterator (viditeľný z hlavného bloku)
  - interná funkcia
- existujúce iterátory:
  `pairs`, `ipairs` (pozri tabuľky); `lines` (pozri súbory)
- vlastné iterátory

vlastný iterátor

Vytvorme iterátor vracajúci údaje len z párnych indexov:

```
function only_at_even_indices(t)
  local i, n = 0, #t
  return function ()
      i = i + 2
      if (i <= n) then return t[i] end
    end
end
```

súbory

súbory a OS

- logické a fyzické hľadisko v súborech
- závislosť od hardvéru (fyzická)
- nezávislosť od hardvéru (logická)
- názvy súborov
- vlastnosti súborov

tri kritériá

- riadiace znaky (textové/binárne)
- spôsoby (režimy) práce
- prístup k údajom

textové a binárne súbory

- podľa použitia riadiacich znakov:
  - textové súbory
  - netextové súbory so zadaným typom údajov
  - netextové súbory bez špecifikovaného typu
- textové súbory ako znakové súbory
- je vnútorne usporiadaný do riadkov
- koniec riadkov (OS)

spracovanie súborov

- súbory len na čítanie
- súbory iba na zápis
- čítanie i zapisovanie do súborov

- vstup vs výstup

spracovanie súboru

"r"	režim iba na čítanie, predvolený režim
"w"	režim povolený zápis; prepíše alebo vytvorí nový súbor
"a"	režim pridávania, ktorý otvorí existujúci súbor alebo vytvorí nový súbor
"r+"	režim čítania a zápisu pre existujúci súbor
"w+"	všetky existujúce údaje sa odstránia, ak súbor existuje, alebo sa vytvorí nový súbor s oprávneniami na čítanie a zápis
"a+"	režim pridávania so zapnutým režimom čítania, ktorý otvorí existujúci súbor alebo vytvorí nový

spracovanie súboru – príklad

```
local f = io.open("myfile.txt", "r") -- pozri vyššie
local words = = f:read("*a")         -- pozri ďalej
```

"*all"	"*a"	prečíta celý súbor
"*ria-dok"	"*l"	prečíta ďalší riadok
"*číslo"	"*n"	číta číslo (vrátane medzier na začiatku)
num		číta reťazec, jeho dĺžka je určená hodnotou *num*

prístup k údajom

- súbory spracovávané postupne
  zvyčajne textové súbory
- súbory s priamym prístupom

súbor existuje?

- žiadna špeciálna funkcia
- vyriešený čítaním nula znakov:

```
if f:read(0) then ...
```

textové súbory

- typický prípad
- lines: funkce riakového iterátoru:
  io.lines(), f:lines()
- zpracovanie CSV

textové soubory – příklad

```
for line in f:lines() do ... end
```

moduly

terminológia

- štandardné knižnice
- užívateľské knižnice
- izolovaný kus kódu
- rozhranie (globálne)
- vnútorné štruktúry (lokálne)
- implementácia funkcií
- inicializačné operácie

príklad vlastnej knižnice

```
local mt = {}                    -- mt = mytriangle

function mt.circumference (a, b, c)
  return a+b+c
end

function mt.area (a, b, c)
  local s = mt.circumference(a,b,c) / 2
  return math.sqrt(s*(s-a)*(s-b)*(s-c))
end

return mt -- important!
```

principy

- metódy (funkcie) patria do hashu
- hash sa vráti
- sú možné aj iné spôsoby

- sa pripája k modulu:

```
local m = require "mytriangle"
io.read(a,b,c)
print(m.circumference(a,b,c), m.area(a,b,c))
```

výhody

- zhromažďuje súvisiace funkcie do jedného celku
- zdieľa kód
- jednoduchšie skladanie nových projektov
- implementácia abstraktných dátových typov

abstrakné dátové typy

vlastnosti

- určuje dátové zložky
- určuje operácie a ich vlastnosti
- odhliada od zvolenej implementácie

výhody

- ADT je určený tým, čo v ňom chceme/potrebujeme
- ADT je možné implementovať rôznymi spôsobmi bez toho, aby to ovplyvnilo jeho správanie
- ADT je implementované pomocou vhodnej dátovej štruktúry (DS)

prehľad ADT

- zásobník (Stack)
- fronta (Queue)
- unikátna množina (Set)
- multimnožina (MultiSet)
- ...

fronta (FIFO)

- FIFO = First-in, First-out
- přístup len k prvku na začiatku (front, head)
- vloženie len na koniec fronty (end, tail)

fronta (diagram signatúry)

- hlavný dátový typ
- súvisejúce dátové typy
- orientované spojnice dátových tokov

- $\Rightarrow$ rozhranie

typické operácie

- konstruktor (init)
- modifikátory (put, get)
- dotazy (size)
- predikáty (empty)

fronta (axiomatický popis)

```
init(_) :          --> queue
count(_): queue --> number
empty(_): queue --> boolean
put(_,_): queue, data --> queue
get(_):   queue --> data
```

fronta (implementácia)

```
local Q = {}

Q.init  = function (t) return t end
Q.put   = function (t,e) table.insert(t,e) end
Q.get   = function (t) local e = table.remove(t,1)
            return e
           end
Q.count = function (t) return #t end
Q.empty = function (t) return #t==0 end
Q.print = function (t)
            for i=1,#t do io.write(t[i]," ") end
            print ()
          end

return Q
```

zásobník (LIFO)

- LIFO = Last-in, First-out
- prístup iba k prvku na začiatku (vrchole)
- vkládanie iba k prvku na začiatku (vrchole)

komunikácia s OS

soubory

- pozri čas o súboroch
- konfigurácia SW: pozri textové súbory
- Linux: predspracovanie pomocou príkazov
- všetky metódy patria do modulu `os`

environmentálne premenné

- čítanie, ale nie je možná žiadna úprava
- pripravuje vlastnú kópiu environmentálnych premenných

```
print (os.getenv("USER"))

local envvars = {}
for envline in io.popen("set"):lines() do
  envname = envline:match("^[^=]+")
  envvars[envname] = os.getenv(envname)
end
```

vykonávanie príkazov

- potrebná znalosť príkazov OS
- function vracia `true` alebo `nul`

```
os.execute("mkdir new_directory")
```

parametre príkazového riadku

- počítajú sa zľava
- #0 = názov spusteného programu
- indexované pole `arg`
- volby/prepínače spolu s parametrami

```
local a, b = arg[1], arg[2]
if #arg>1 then
  print (a+b)
end
```

iné

- dátum+čas

```
print(os.date("today is %A, %B %d"))
Today is Monday, September 15
print(os.time("Now i
```

aplikácie

LuaTEX

- počítačový sádzací systém založený na TEXe, ktorý začal ako verzia pdfTEX so zabudovaným skriptovacím strojom Lua
- vnútorné štruktúry prístupné cez Lua

Lua v ConTEXt

- ConTEXt: rozšírenie základného TEXu
- vostavený intrepret Lua
- Lua umožňuje náročnešie operácie, ktoré sú zložitejšie pomocou nástrojov TEX alebo ConTEXt
- tlač (funkcia `context`) do výstupného prúdu (PDF)
- funkcia inspect implementována

Lua v ConTEXt – príklad

```
\startluacode
a=math.sqrt(2)
context(a)
\stopluacode
...
\ctxlua{ .. commands .. }
```

hry

- prečo Lua: kompilátor a interpret môže byť jednoducho vložený do ľubovoľného aplikácie. (Alebo len jeho časť.)
- framework pre 2D (napr. LÖVE, Pygame)
- frameworky pre 3D (napr. Pyglet)
- vzťahy s inými knižnicami (napr. OpenGL)

Programovací jazyk Lua

Tomáš Hála

Učebný materiál

Funded by
the European Union