

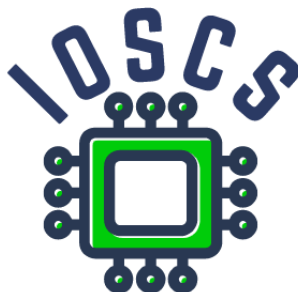
**Project: Innovative Open Source Courses for Computer Science**

# **Programovací jazyk Lua** **Učební materiál**

**Tomáš Hála**  
**Mendel University in Brno**

**29. 5. 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

#### Erasmus+ Disclaimer

This project has been funded with support from the European Commission. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

#### Copyright Notice

This content was created by the IOSCS consortium: 2019–2022. The content is Copyrighted and distributed under Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0).



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

# Programovací jazyk Lua

Tomáš Hála

Učební materiál



Funded by  
the European Union

# úvod

# algoritmy – vlastnosti

---

- jednoznačný (deterministický)
- konečný, tzn. vždy vede k určitým výsledkům
- obecný, tzn. použitelné pro řešení daného problému s využitím jakýchkoliv přípustných údajů
- opakovatelný, tzn. vždy vede ke stejným výsledkům se stejnými vstupními daty

# algoritmy – vyjadřující

---

- verbálně – v přirozeném jazyce
- graficky – vývojový diagram nebo strukturogram
- matematicky – vztah mezi veličinami, soustava rovnic, matice
- programovací jazyk

# algoritmizace

---

- input: problém
- výstup: algoritmus

# programování

---

- vyjádření algoritmu
- programovací jazyk(y)
- ladění
- testování
- vstupní data, výstupní informace



# programovací jazyky

---

- program v programovacím jazyce: čitelný pro člověka, ale počítač tomu nerozumí
- strojový kód
- kompilace, kompilátor
- interpretované programy, interprety

# o jazyce Lua

# historie

---

- 1993

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

Computer Graphics Technology Group (Tecgraf), Pontifical Catholic University of Rio de Janeiro, Brazil

- multiparadigmatický:

- skriptovací

- imperativní (procedurální, prototypovací, objektově orientovaný)

- funkcionální

# versions

---

- 5.1.x
- 5.2.x
- 5.3.x (zde použítá)
- 5.4.3 (poslední)

# 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://geekflare.com/online-compiler/lua)

[\*\*https://www.jdoodle.com/execute-lua-online/\*\*](https://www.jdoodle.com/execute-lua-online/)

[\*\*https://onecompiler.com/lua/3y5j9aajb\*\*](https://onecompiler.com/lua/3y5j9aajb)

[\*\*https://replit.com/languages/lua\*\*](https://replit.com/languages/lua)

[\*\*https://www.tutorialspoint.com/execute\\_lua\\_online.php\*\*](https://www.tutorialspoint.com/execute_lua_online.php)

[\*\*https://www.lua.org/demo.html\*\*](https://www.lua.org/demo.html)

# struktura jazyka

# struktura – 22 vyhrazených slov

---

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

konstanty	false true nil
proměnné	local
operátory	and not or
větvení	if then else elseif end
cykly	for in repeat until while do end
funkce	function return
skoky	break goto



# větvení – 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
```

Příklad:

```
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
```

# datové typy a operátory

# obecně

---

- každá proměnná v programu je určitého typu
- každý programovací jazyk definuje vlastní množinu datových typů

# datový typ tvoří definice:

---

- množiny hodnot
- vnitřní reprezentace hodnot v počítači (velikost v paměti, kódování hodnot)
- množiny přípustných hodnot



# srovnej:

---

- Např. (Pascal) `var x:boolean;`
  - hodnoty: `true`, `false`
  - zábor: 1 byte, bit 0 je rozhodující
  - operace: `not`, `and`, `or`

# v jazyce Lua

---

- dynamicky typovaný jazyk
- nejsou definice typů
- hodnoty nesou informaci o příslušnosti k datovému typu
- osm základních datových typů:
  - nil
  - boolean, number, string
  - function
  - table
  - userdata, thread (zde neprobíráme)

# nil

---

- nil je nil :-)
- liší se všech ostatních hodnot
- nepřítomnost užitečné hodnoty

# boolean

---

- false a true

# number

---

- pro celočíselné hodnoty i hodnoty s plovoucí řádovou čárkou
- 8 B
- největší hodnota?

# string

---

- immutabilní posloupnost bytů
- řetězec může obsahovat jakoukoliv osmibitovou hodnotu, včetně `'\0'`
- nepředpokládá se žádné kódování

# number – aritmetické operace

---

+	sčítání
–	odčítání
*	násobení
/	dělení
//	celočíselné dělení
%	modulo (zbytek po dělení)
^	umocňování
–	unární minus

# number – bitové operace

---

- ~ unární bitové NOT
- & bitové AND
- | bitové OR
- ^ bitové exkluzivní OR
- >> bitový posun doprava
- << bitový posun doleva



# boolean – logické operace

---

- not
- and or
  
- obvyklý význam
- používáno též u ostatních typů

# string – spojení, délka

---

- ..
  - čísla se konvertují na string
  - #
  - number of *bytes*
- 
- # viz tabulky

# relační operátory

---

$==$	rovnost
$\neq$	nerovnost
$<$	menší než
$\leq$	menší nebo rovno
$>$	větší než
$\geq$	větší nebo rovno

# priorita operátorů (od nejvyšší po nejnižší)

---

^

unární operátory (not # - ~)

\* / // %

+ -

..

<< >>

&

~

|

< > <= >= ~= ==

and

or

funkce

# terminologie

---

- funkce vs. procedury
- deklarace (hlavička funkce)
- parametry (nelze použít pro úpravu hodnot v hlavním bloku)
- návratová hodnota(y)

# terminologie

---

- předdefinované vs. vlastní funkce
- body: doporučují se lokální proměnné
- zvláštní druh: iterační funkce (vysvětleno bude později)

# funkce – příklad

---

```
funkce obvod_trojuhelnik (a, b, c)
    vrátit a+b+c
konec
```

## S návratovou hodnotou

```
funkce myprint (a, b, c)
    print("hodnota a je", a)
    print("hodnota b je", b)
    print("hodnota c je", c)
konec
```

## Bez návratové hodnoty



# rekurzivní funkce

---

- volají samy sebe
- každá definice rekurzivního algoritmu musí obsahovat hodnotu pro ukončení rekurze (hodnota 1 v následujícím příkladu)
- efektivní?

# rekurzivní funkce

---

- přímá rekurze: volá se přímo
- nepřímá rekurze: nutné dvě nebo více funkcí – F1 volá F2 a F2 volá F1

# rekurzivní funkce – příklad

---

```
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 programu:

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

# řetězce

# nástroje

---

řetězec je objekt  
metody:

```
.. -- zřetězení  
string.len(arg) -- délka  
string.rep(s, n) -- replikuje řetězec n-krát
```

```
řetězec.horní  
string.lower(s)
```

```
string.reverse(y)  
string.char(x) -- obj. hodnota --> znak/řetězec  
string.byte(ch) -- char --> ord. hodnota
```

# formátování výstupu

---

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

# nástroje: hledat & nahradit

---

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

Vrátí řetězec s nahrazením výskytů `fs` za `rs`.

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

Vrátí fragmenty `s` popsané pomocí `pattern`.

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

Vrátí počáteční a koncový index `fs` v `s` (nebo `nil`, pokud nebyl nalezen).

# nástroje: hledat & nahradit

---

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

startindex je i-tý index endindex je j-tý index posledního indexu řetězce, který chceme

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



# chybějící nástroje

---

- `split` například pro rozdělení dat ve formátu CSV
- `trim` pro odstranění koncových mezer
- můžeme psát vlastní funkce

# vzory v jazyce Lua

---

Regulární výraz POSIX vs. vzory Lua

# vzory v jazyce Lua

---

třídy:

- . všechny znaky
- %a písmena
- %c řídicí znaky
- %d číslice
- %l malá písmena
- %p interpunkční znaménka
- %s mezery
- %u velká písmena
- %w alfanumerické znaky
- %x hexadecimální číslice
- %z znak s ordinální hodnotou 0

# vzory v jazyce Lua

---

## doplňky množiny

- `%A` písmen
- `%C` řídicích znaků
- `%D` číslic
- `%L` malých písmen
- `%P` interpunkčních znaků
- `%S` mezer
- `%U` velkých písmen
- `%A` alfanumerických znaků
- `%X` hexadecimálních číslic
- `%Z` znaků s ordinální hodnotou 0

# vzory v jazyce Lua

---

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

%

^ \$

+ - \* ?

[ ] ( ).

# split

---

## kód funkce 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 --- zjednodušená verze pouze pro neprázdná pole
```

# rozdělit

---

verze pro zachování prázdných polí:

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

# split – použití

---

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



# split – použití

---

špinavý trik:

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

[možné, ale nedoporučuje se]

# strukturované datové typy

# obecně

---

- (indexované) pole
- záznam / struct
- bitové pole (set)
- asociativní pole (hash)
  
- objekt
  
- A co v Lua?

# V jazyce Lua:

---

**table**

# konstruktory

---

```
t={}
```

```
t[1]=1
```

```
t[2]=2
```

```
t[3]=7
```

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

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

= homogenní pole, indexované

# konstruktory II.

---

t={}

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

= heterogenní pole, indexované

# konstruktory III.

---

```
t={}
```

```
t["jan"]=31
```

```
t["feb"]=28
```

```
t["mar"]=31
```

```
t.jan=31
```

```
t.feb=28
```

```
t.mar=31
```

= asociativní pole (hash)

# konstruktory IV.

---

```
m="jan"
```

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

```
...
```

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

... kombinace indexovaného a asociativního pole



# knihovna table

---

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

# knihovna table II.

---

```
t={}  
table.insert(t,"Monday")  
table.insert(t,"Tuesday")  
table.insert(t,"Wednesday")  
  
for i=1,#t do  
    print(t[i])  
end
```

# tabulka – výstup

---

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

# převod pole na hash

---

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

```
h = {}
```

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

výsledkem je množina

# převod pole 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
```

výsledkem je multimnožina

# vzestupně, sestupně, nebo...?

---

```
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  
)
```

funkce II.

# funkce jako datový 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))
```



# funkce jako datový 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)
```

# funkce: iterátory a uzávěry

---

- iterující funkce postupně procházejí data (tabulky, soubory)
- dvě funkce:
  - iterátor (viditelný z nadřízeného bloku)
  - vnitřní funkce
- existující iterátory:  
pairs, ipairs (viz tabulky); lines (viz soubory)
- vlastní iterátory

# vlastní iterátor

---

Vytvořme iterátor vracející data pouze ze sudých indexů:

```
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
```

soubory

# soubory a OS

---

- logické a fyzické hledisko v souborech
- závislost na hardwaru (fyzická)
- nezávislost na hardwaru (logická)
- názvy souborů
- vlastnosti souboru

# tři kritéria

---

- řídicí znaky (text/binární)
- způsoby (režimy) práce
- přístup k datům

# textové a binární soubory

---

- podle použití řídicích znaků:
  - textové soubory
  - netextové soubory se shodným datovým typem
  - netextové soubory s daty různých datových typů
- textové soubory jako znakové soubory
- vnitřní uspořádání do řádků
- konec řádků (OS)

# zpracování souborů

---

- soubory pouze pro čtení
- soubory pouze pro zápis
- čtení i zápis do souboru
  
- vstup vs. výstup



# zpracování souborů

---

- "r" pouze pro čtení, výchozí režim
- "w" povolený zápis; přepíše nebo vytvoří nový soubor
- "a" přidává k obsahu existujícího souboru nebo vytvoří nový soubor
- "r+" čtení i zápis v existujícím souboru
- "w+" všechna existující data jsou odstraněna, pokud soubor existuje, nebo je vytvořen nový soubor s oprávněním pro čtení a zápis
- "a+" přidávání k existujícímu souboru, nebo vytvoření nového souboru

# zpracování souborů – příklad

---

```
local f      = io.open("myfile.txt", "r")  -- see above
local words = f:read("*a")                -- see below
```

# přístup k datům

---

- soubory zpracovávané postupně  
obvykle textové soubory
- soubory s přímým přístupem

# soubor existuje?

---

- žádná speciální funkce
- vyřešen čtením nulového počtu znaků:

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

# textové soubory

---

- typický případ
- lines: funkce řádkového iterátoru:  
io.lines(), f:lines()
- zpracování CSV

# text files – example

---

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

moduly

# terminologie

---

- standardní knihovny
- uživatelské knihovny
- izolovaný kus kódu
- rozhraní (globální)
- vnitřní struktury (lokální)
- implementace funkcí
- inicializační operace



# příklad vlastní knihovny

---

```
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

---

- metody (funkce) se vkládají do asociativního pole
- toto pole je vráceno
- jsou možné i jiné způsoby
  
- připojení modulu:

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

# výhody

---

- shromažďuje související funkce do jednoho celku
- sdílí kód
- jednodušší skládání nových projektů
- implementace abstraktních datových typů

# abstraktní datové typy

# vlastnosti

---

- určuje datové složky
- určuje operace a jejich vlastnosti
- odhlíží od zvolené implementace

# výhody

---

- ADT je určen tím, co v něm chceme/potřebujeme.
- ADT lze implementovat různými způsoby, aniž by to ovlivnilo jeho chování
- ADT je implementováno pomocí vhodné datové struktury (DS)

# přehled ADT

---

- Zásobník (Stack)
- Fronta (Queue)
- Množina (Set)
- Množina s opakováním  
(Multimnožina; MultiSet)
- ...

# fronta (FIFO)

---

- FIFO = First-in, First-out
- přístup pouze k prvku na začátku (front, head)
- vložení pouze na konec fronty (end, tail)



# fronta (diagram signatury)

---

- hlavní datový typ
- související datové typy
- orientované spojnice datových toků
  
- ⇒ rozhraní

# typické operace

---

- 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 (implementace)

---

```
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
- přístup pouze k prvku na začátku (vrcholu)
- vložení pouze na začátek (vrchol)

# komunikace s OS

# soubory

---

- diskutováno dříve
- konfigurace SW: viz textové soubory
- Linux: předzpracování pomocí příkazů
- všechny metody patří do modulu os

# proměnné prostředí

---

- čtení, ale není možná žádná úprava
- připravuje vlastní kopii proměnných prostředí

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

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



# provádění příkazů

---

- nutná znalost příkazů OS Funkce
- vrací true nebo nil

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

# parametry příkazového řádku

---

- počítané zleva
- #0 = název běžícího programu
- indexované pole arg
- volby/přepínače zároveň s parametry

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

## ■ datum+čas

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

aplikace

- počítačový sázecí systém založený na TeXu, který začal jako verze pdfTeX se zabudovaným skriptovacím strojem Lua
- vnitřní struktury přístupné přes struktury jazyka Lua

# Lua v ConT<sub>E</sub>Xtu

---

- ConT<sub>E</sub>Xt: rozšíření základního T<sub>E</sub>Xu
- vestavěný interpret jazyka Lua
- Lua umožňuje náročnější operace, které jsou složité pomocí nástrojů T<sub>E</sub>X nebo ConT<sub>E</sub>Xt
- tisk (funkce `context`) do výstupního proudu (PDF)
- funkce `inspect` implementována

# Lua v ConT<sub>E</sub>Xtu – příklad

---

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

# hry

---

- proč Lua: kompilátor a interpret lze jednoduše vložit do libovolného aplikace. (Nebo jen jejich část.)
- frameworky pro 2D (např. LÖVE, Pygame)
- frameworky pro 3D (např. Pyglet)
- vztahy k jiným knihovnám (např. OpenGL)



# Programovací jazyk Lua

Tomáš Hála

Učební materiál



Funded by  
the European Union