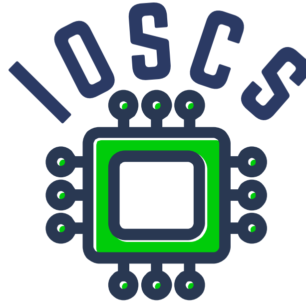
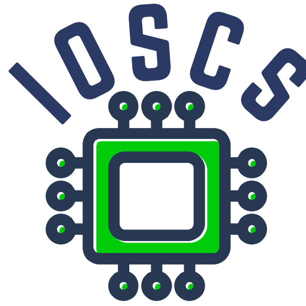


Project: Innovative Open Source Courses for Computer Science

Algorithmisation and Programming in Lua
Teaching Material

Tomáš Hála
Mendel University in Brno

29. 8. 2020



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 BRNĚ
ŽILINSKÁ UNIVERZITA V ŽILINE

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 licence (CC BY-SA 4.0).

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

Język programowania Lua

Tomáš Hála

Materiał naukowy

Funded by
the European Union

Wstęp

algorytmy – właściwości

- jednoznaczne (deterministyczne)
- skończony (wynikowy), tj. zawsze prowadzi do pewnych rezultatów
- ogólnie, tj. mający zastosowanie do rozwiązania danego problemu przy użyciu dowolnych dopuszczalnych danych
- powtarzalny, tj. zawsze prowadzi do tych samych wyników przy tych samych danych wejściowych

algorytmy – wyrażanie

- ustnie – w języku naturalnym
- graficznie – schemat blokowy lub schemat struktury
- matematycznie – związek między wielkościami, układ macierzy równań
- językiem programowania

algorytmizacja

- wejście: problem
- wyjście: algorytm

programowanie

- wyrażenie algorytmu
- języki programowania
- debugowanie
- testowanie
- dane wejściowe, informacje wyjściowe

języki programowania

- program w języku programowania: czytelny dla człowieka ale komputer tego nie rozumie
- kod maszynowy
- kompilacja, kompilator
- interpretowane programy, tłumacze

o Lua

historia

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

  Computer Graphics Technology Group (Tecgraf), Pontifical Catholic University of Rio de Janeiro, Brazylia
- wieloparadygmatyczny:
  - język skryptowy
  - imperatywny (proceduralny, prototypowy, obiektowo zorientowany)
  - funkcjonalny

wersje

- 5.1.x
- 5.2.x
- 5.3.x (używane tutaj)
- 5.4.3 (najnowsza)

źródła

```
 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

struktura języka

22 zastrzeżone słowa

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

stałe	`false true nil`
zmienny	`local`
operatorzy	`and not or`
rozgałęzienie	`if then else elseif end`
cykle	`for in repeat until while`
	`do end`
funkcje	`function return`
skoki	`break goto`

polecenie rozgałęzienia – if

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

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

cykle – while

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

Przykład:

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

cykle – repeat

```
repeat
 ...
until condition

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

cykle – for I.

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

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

cykle – for II.

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

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

typy danych a operatory

w ogóle

- każda wartość przetwarzana w programie jest jakiegoś typu
- każdy język programowania określa swój własny zestaw typów użytkowych

typ danych zawiera:

- zbiór wartości
- wewnętrzną reprezentację w komputerze (rozmiar pamięci, kodowanie wartości)
- zbiór operacji dozwolonych dla danego typu

porównaj:

- Np: (Pascal) `var x:boolean;`
  - wartośc true i false
  - wymaga 1 bajtu, bit 0 jest znaczący
  - operacje: not, and, or

W Lua

- język dynamicznie typowany
- brak definicji typów w języku
- wartości posiadają swój własny typ
- osiem podstawowych typów:
  - nil
  - boolean, number, string
  - function
  - table
  - userdata, thread (nie omawiane tutaj)

nil

- nil to nil :-)
- różni się od każdej innej wartości
- brakiem użytecznej wartości

boolean

- false i true

number

- zarówno dla liczb całkowitych, jak i zmiennoprzecinkowych
- 8 B
- największa wartość?

string

- niezmienne sekwencje bajtów
- ciąg mogą zawierać dowolne wartości 8-bitowe, w tym ’\0’
- brak założeń dotyczących kodowania

number – operatory arytmetyczne

+	dodawanie
–	odejmowanie
*	mnożenie
/	dzielenie
//	dzielenie całkowite
%	reszta po dzieleniu
^	wykładanie
–	unarny minus

number – operacje bitowe

~	bitowe nie (inwersja) NOT
&	bitowe AND
|	bitowe OR
~	bitowe ekskluzywne OR
>>	przesunięcie bitowe w prawo
<<	przesunięcie bitowe w lewo

boolean – operatory logiczne

- not
- and, or

- zwykłe znaczenie
- używane również dla innych typów danych

string – konkatenacja, długość

- ..
- liczby są konwertowane na łańcuchy
- #
- liczba *bajtów*

- # patrz tabele

operatory relacyjne

==	równość
~=	nierówność
<	mniejsza niż
<=	mniejsza lub równa
>	większa niż
>=	większy lub równy

priorytet (od wyższego do niższego)

```
^
unarne operatory (not   #     -     ~)
*     /     //    %
+     -
..
<<    >>
&
~
|
<     >     <=    >=    ~=    ==
and
or
```

funkcje

terminologia

- funkcje v procedury
- deklaracja (głowa funkcji)
- parametry (nie mogą być używane do modyfikowania wartości w głównym bloku)
- zwracane wartości

terminologia

- predefiniowane v własne funkcje
- body: zalecane zmienne lokalne
- specyficzny rodzaj: funkcje iteracyjne (wyjaśnione później)

funkcje – przykład

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

Z wartością zwracaną

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

Bez zwracanej wartości

funkcje rekurencyjne

- wywołujący sam siebie
- każda definicja algorytmu rekurencyjnego musi zawierać wartość for zakończenie rekurencji (wartość 1 w poniższym przykładzie)
- skuteczny?

funkcje rekurencyjne

- bezpośrednia rekurencja: bezpośrednie wywołanie samego siebie
- rekurencja pośrednia: konieczne dwie lub więcej funkcji –
  F1 wywołuje F2, a F2 wywołuje F1

funkcje rekurencyjne — przykład

```
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

-- w programe:

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

ciągi znaków

narzędzia

string jest obiektem
metody:

```
.. -- konkatenacja
string.len(arg) -- długość
string.rep(s, n)) - replikuje łańcuch n-krotnie

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

string.reverse(s)
string.char(x) -- ord. wartość --> znak/łańcuch
string.byte(ch) -- char --> ord. wartość
```

formatowanie wyjścia

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

narzędzia: wyszukaj & zamień

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

Zwraca łańcuch, zastępując wystąpienia `fs` przez `rs`.

`string.gmatch( s, wzór)`

Zwraca fragmenty s opisane przez `pattern`.

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

Zwraca indeks początkowy i końcowy `fs` w `s` (lub `nil`, jeśli nie znaleziono).

narzędzia: wyszukaj & zamień

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

startindex jest i-tym indeksem endindex jest j-tym indeksem ostatniego indeksu żądanego fragmentu łańcucha

```
s = "To jest mój tekst."
print(string.sub(s, 2, 3))
print(string.sub(s, 2, -2))
```

brakujące narzędzia

- `split` do dzielenia np. danych CSV
- `trim` do wycinania końcowych spacji

- możemy napisać własne funkcje

wzorce Lua

Wyrażenia regularne POSIX
versus wzorce Lua

wzorce Lua

zajęcia:

```
. wszystkie postacie
% liter
%c znaków sterujących
%d cyfr
%l małe litery
%p znaków interpunkcyjnych
%s znaków spacji
%u wielkie litery
%w znaków alfanumerycznych
%x cyfr szesnastkowych
%z znak z reprezentacją 0[]
```

wzorce Lua

komplementy do:

```
%A liter
%C znaków sterujących
%D cyfr
%L małe litery
%P znaków interpunkcyjnych
%S znaków spacji
%U wielkie litery
%A znaków alfanumerycznych
%X cyfr szesnastkowych
%Z znak z reprezentacją 0
```

Lua patterns

escape, kotwice, iteratory (modyfikatory), zbióry + grupy:

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

split

kod funkcji mysplit (tylko dla niepustych pól)

```
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
```

split

wersja dla pustych pól:

```
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
```

split – użyj

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

split – użyj

„dirty trik”

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

[możliwe, ale niezalecane]

strukturyzowane typy danych

ogólnie

- (indeksowana) tablica
- rekord / struktura
- tablica bitowa (zestaw)
- tablica asocjacyjna (hash)

- objekt

- A co z 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,}
```

= jednorodna tablica, indeksowana

konstruktory II.

```
t={}
t={1,2,7,"Lua",true,{},-9,9999,false,8888}
```

= tablica heterogeniczna, indeksowana

konstruktory III.

```
t={}

t["sty"]=31
t["luty"]=28
t["mar"]=31

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

= tablica asocjacyjna (hash)

konstruktory IV.

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

t={january=31,february=28,mar=31}
```

... połączenie tablicy indeksowanej i tablicy mieszającej
Pole

table biblioteka

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

table biblioteka II.

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

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

tabela – dane wyjściowe

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

table do hash

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

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

wynikiem jest zestaw

table do 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
```

wynikiem jest multiset

rosnąco, malejąco, czy...?

```
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
        )
```

funkcje II.

funkcja jako typ danych

```
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))
```

funkcja jako typ danych

```
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)
```

funkcje: iteratory i domknięcia

- funkcja iteracyjna umożliwia przechodzenie przez dane (tabele, pliki)
- dwie funkcje:
  - iterator (widoczny z głównego zakresu)
  - funkcja wewnętrzna
- istniejące iteratory:
  `pary`, `ipary` (patrz tabele); `lines` (zobacz pliki)
- własne iteratory

własny iterator

Stwórzmy iterator zwracający dane tylko z parzystych indeksów:

```
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
```

pliki

pliki i system operacyjny

- logiczny i fizyczny punkt widzenia w plikach
- zależność od sprzętu (fizycznego)
- niezależność od sprzętu (logiczna)
- nazwy plików
- właściwości pliku

trzy kryteria

- znaki kontrolne (tekstowe/binarne)
- obsługa (tryby)
- dostęp do danych

pliki tekstowe i binarne

- zgodnie z użyciem znaków kontrolnych:
  - pliki tekstowe
  - pliki nietekstowe o określonym typie danych
  - pliki nietekstowe bez określonego typu
- pliki tekstowe jako pliki znakowe
- wewnętrznie zorganizowane w linie
- koniec linii (OS)

obsługa plików

- pliki tylko do odczytu
- pliki tylko do zapisu
- zarówno odczytuje, jak i zapisuje pliki

- wejście v wyjście

obsługa plików

"r"	tryb tylko do odczytu, tryb domyślny
"w"	tryb zezwolenia na zapis; nadpisuje lub tworzy nowy plik
"a"	tryb dołączania, który otwiera istniejący plik lub tworzy nowy plik do dołączenia
"r+"	tryb odczytu i zapisu istniejącego pliku
"w+"	wszystkie istniejące dane są usuwane, jeśli plik istnieje lub tworzony jest nowy plik z uprawnieniami do odczytu i zapisu
"a+"	tryb dołączania z włączonym trybem odczytu, który otwiera istniejący plik lub tworzy nowy plik

obsługa plików – przykład

```
local f = io.open("myfile.txt", "r") -- patrz wyżej
local words = f:read("*a") -- patrz poniżej
```

"*all"	"*a"	odczytuje cały plik
"*lines"	"*l"	czyta następną linię
"*number"	"*n"	odczytuje liczbę (wraz z białymi znakami)
num		czyta ciąg znaków, jego długość jest określona przez wartości num

dostęp do danych

- pliki przetwarzane sekwencyjnie
  zazwyczaj pliki tekstowe
- pliki z bezpośrednim dostępem

plik istnieje?

- nie ma specjalnej funkcji
- rozwiązany przez odczytanie zerowych znaków:

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

pliki tekstowe

- typowy przypadek
- linie: funkcje iteratora linii: io.lines(), f:lines()
- przetwarzanie pliku CSV

pliki tekstowe – przykład

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

moduły

terminologia

- standardowe biblioteki
- biblioteki użytkownika
- izolowany fragment kodu
- interfejs (globalny)
- struktury wewnętrzne (lokalne)
- implementacja funkcji
- operacje inicjujące

przykład własnej biblioteki

```
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!
```

zasady

- metody (funkcje) należą do skrótu
- hash jest zwracany
- możliwe są też inne sposoby

- łączący moduł:

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

zalety

- łącząc powiązane funkcje w jedną całość
- udostępnia kod
- łatwiejsze komponowanie nowych projektów
- implementacja abstrakcyjnych typów danych

abstrakcyjne typy danych

nieruchomości

- określać wykorzystywane składniki danych
- określać operacje i ich właściwości
- abstrahować od metody implementacji

korzyści

- ADT jest zdeterminowane przez to, czego w nim chcemy/potrzebujemy
- ADT może być zaimplementowane na różne sposoby bez wpływu na jego zachowanie
- ADT jest zaimplementowany przy użyciu odpowiedniej struktury danych (DS)

przegląd ADT

- stos (Stack)
- kolejka (Queue)
- zbiór (Set)
- multizbiór (MultiSet)
- ...

kolejka (FIFO)

- FIFO = First-in, First-out
- dostęp tylko do elementu na początku (front, head)
- wstawianie tylko na koniec kolejki (end, tail)

kolejka (schemat sygnatury)

- główny typ danych
- powiązane typy danych
- strzałki przepływu danych

- $\Rightarrow$ interfejs

typowe operacje

- konstruktor (init)
- modyfikatory (wstaw, pobierz; put, get)
- zapytania (rozmiar; size)
- predykaty (puste; empty)

kolejka (opis aksjomatyczny)

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

kolejka (implementacja)

```
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
```

stos (LIFO)

- LIFO = Last-in, First-out
- dostęp tylko do elementu na górze
- wstawianie tylko na górze

komunikacja z OS

pliki

- omówiony wcześniej
- konfiguracja SW: patrz pliki tekstowe
- Linux: wstępne przetwarzanie za pomocą poleceń
- wszystkie metody należą do modułu `os`

zmienne środowiskowe

- odczyt, ale modyfikacja nie jest możliwa
- przygotowuje własną kopię zmiennych środowiskowych

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

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

wykonywanie poleceń

- wymagana znajomość poleceń systemu operacyjnego
- funkcja zwraca `true` lub `nil`

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

parametry linii poleceń

- wyliczany od lewej
- #0 = nazwa uruchomionego programu
- tablica indeksowana arg
- opcje wraz z parametrami

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

inne

- data+godzina

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

aplikacje

LuaTeX

- komputerowy system składu oparty na TeX-u, który powstał jako wersja pdfTeX z wbudowanym silnikiem skryptowym Lua.
- internals dostępne przez Lua

Lua w ConTEXt

- ConTEXt: rozszerzenie podstawowego TEX-a
- embedded Lua zinterpretowane
- Lua umożliwia bardziej wyrafinowane operacje, które są bardziej skomplikowane za pomocą narzędzi TEX lub ConTEXt
- drukowanie (funkcja `context`) do strumienia wyjściowego (PDF)
- inspect zaimplementowano

Lua w ConTEXt – przykład

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

gry

- dlaczego Lua: kompilator i interpreter można po prostu osadzić w dowolnym wniosek. (Lub tylko ich część.)
- frameworki dla 2D (np. LÖVE, Pygame)
- frameworki dla 3D (np. Pyglet)
- relacje z innymi bibliotekami (np. OpenGL)

Język programowania Lua

Tomáš Hála

Materiał naukowy

Funded by
the European Union