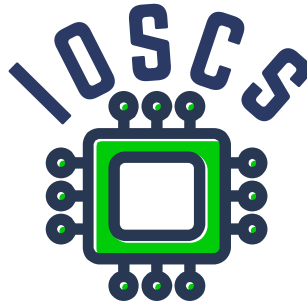


Project: Innovative Open Source Courses for Computer Science

# 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 IOCS consortium: 2019–2022. The content is copyrighted and distributed under Creative Commons Attribution-ShareAlike 4.0 International licence (CC BY-SA 4.0).

# Programming language Lua

Tomáš Hála

Teaching Material



Funded by  
the European Union

# introduction

# algorithms – properties

---

- unambiguous (deterministic)
- finite (resultative), ie. always leading to certain results
- general, ie. applicable to the solution of a given problem using any admissible data
- repeatable, ie. always leading to the same results with the same input data

# algorithms – expressing

---

- verbally – in natural language
- graphically – flowchart or structure chart
- mathematically – a relationship between quantities, a system of equations matrices
- programming language

# algorithmisation

---

- input: problem
- output: algorithm

# programming

---

- expression of algorithm
- programming language(s)
- debugging
- testing
- input data, output information



# programming languages

---

- programme in programming language: human readable but computer does not understand it
- machine code
- compilation, compiler
- interpreted programmes, interpreters

# about Lua

# history

---

- 1993

- Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, *Computer Graphics*

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

- multi-paradigm:

- scripting

- imperative (procedural, prototype-based, object-oriented)

- functional

# versions

---

- 5.1.x
- 5.2.x
- 5.3.x
- 5.4.4 (last)

# sources

---

```
sudo 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.tutorialspoint.com/execute_lua_online.php)

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

# structure of the language

# structure - 22 reserved words

---

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

constants	false true nil
variables	local
operators	and not or
conditions	if then else elseif end
loops	for in repeat until while do end
functions	function return
jumps	break goto



# conditions – if

---

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

```
if      condition1 then ...  
elseif condition2 then ...  
elseif conditino3 then ...  
else ...  
end
```

# loops – while

---

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

Example:

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

# loops – repeat

---

```
repeat
```

```
...
```

```
until condition
```

```
x,i = 1,5
```

```
repeat
```

```
    x=x*i
```

```
    i=i-1
```

```
until i==1
```

```
print (x)
```

# loops – for I.

---

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

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

# loops – for II.

---

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

```
  ...
```

```
end
```

```
for i=-10,10,2 do
```

```
  print (i, i*2, i^3)
```

```
end
```

# data types & operators

# in general

---

- each value processed in the programme is of one of some type
- each programming language determines its own set of usable types

# data type contains:

---

- the set of values
- the internal representation in the computer (memory size, encoding of values) the
- the set of operations allowed for the type



# compare:

---

- Eg: (Pascal) `var x:boolean;`
  - values true and false
  - needs 1 byte, bit 0 is significant
  - operations: not, and, or

# in Lua

---

- dynamically typed language
- no type definitions in the language
- values carry their own type
- eight basic types:
  - boolean, number, string,
  - function,
  - table,
  - userdata, thread (not discussed here)

# nil

---

- nil is nil :-)
- differs from any other value
- the absence of a useful value.

# boolean

---

- false and true

# number

---

- for both integer numbers and floating-point numbers
- 8 B
- the biggest value?

# string

---

- immutable sequences of bytes
- strings can contain any 8-bit value, including `'\0'`
- no encoding assumptions

# number – arithmetic operators

---

+	addition
–	subtraction
*	multiplication
/	float division
//	floor division
%	modulo
^	exponentiation
–	unary minus

# number – bitwise operators

---

~	unary bitwise NOT
&	bitwise AND
	bitwise OR
^	bitwise exclusive OR
>>	right shift
<<	left shift



# boolean – logical operators

---

- not
- and or
  
- usual meaning
- also used for other data types

# string – concatenation, length

---

- ..
  - numbers are converted to string
  - #
  - number of *bytes*
- 
- *for #, see tables*

# relational operators

---

$==$	equality
$\neq$	inequality
$<$	less than
$\leq$	less or equal
$>$	greater than
$\geq$	greater or equal

# precedence (from higher to lower priority)

---

^

unary operators (not # - ~)

\* / // %

+ -

..

<< >>

&

~

|

< > <= >= ~= ==

and

or

# functions

# terminology

---

- functions v procedures
- declaration (head of a function)
- parameters (cannot be used for modifying values in the main block)
- return value(s)

# terminology

---

- predefined v own functions
- body: local variables recommended
- specific kind: iterating functions (explained later)

# functions – example

---

With the return value

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

Without a return value



# recursive functions

---

- calling itself
- every definition of a recursive algorithm must contain a value for ending the recursion (value 1 in the following example)
- effective?

# recursive functions

---

- direct recursion: calling itself directly
- indirect recursion: two or more functions necessary –  
F1 calls F2 and F2 call F1

# recursive functions – example

---

```
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
```

-- in the programme:

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

# strings

# tools

---

string is an object  
methods:

```
..                -- concatenation
string.len(arg)   -- length
string.rep(s, n)  -- replicates string n-times

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

string.reverse(s)
string.char(x)    -- ord. value --> char/string
string.byte(ch)  -- char --> ord. value
```

# formatting the output

---

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

# tools: search & replace

---

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

Returns a string by replacing occurrences of `fs` with `rs`.

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

Returns fragments of `s` described by `pattern`.

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

Returns the start index and end index of the `fs` in the `s` (or `nil` if not found).

# tools: search & replace

---

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

startindex is the i-th index endIndex is the j-th index of the last index of the string piece we want

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



# missing tools

---

- `split` for splitting eg CSV data
- `trim` for cutting out trailing spaces
- we can write own functions

# Lua patterns

---

POSIX regular expression v Lua patterns

# Lua patterns

---

classes:

. all characters  
%a letters  
%c control characters  
%d digits  
%l lower case letters  
%p punctuation characters  
%s space characters  
%u upper case letters  
%w alphanumeric characters  
%x hexadecimal digits  
%z the character with representation 0 []

# Lua patterns

---

complements to:

`%A` letters  
`%C` control characters  
`%D` digits  
`%L` lower case letters  
`%P` punctuation characters  
`%S` space characters  
`%U` upper case letters  
`%A` alphanumeric characters  
`%X` hexadecimal digits  
`%Z` the character with representation 0

# Lua patterns

---

escape, anchors, iterators (modifiers), sets + groups:

%

^ \$

+ - \* ?

[ ] ( ) .

# split

---

## code of the mysplit function

```
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 -- simplified version for non-empty fields only
```

# split

---

version for empty fields:

```
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 – use

---

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



# split – use

---

dirty trick:

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

[possible but not recommended]

# structured data types

# in general

---

- (indexed) array
- record / struct
- bitwise array (set)
- associative array (hash)
  
- object
  
- What about in Lua?

# table

# constructors

---

```
t={}
```

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

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

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

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

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

= homogeneous array, indexed

# constructors II.

---

```
t={}
```

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

= heterogeneous array, indexed

# constructors III.

---

```
t={}
```

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

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

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

```
t.jan=31
```

```
t.feb=28
```

```
t.mar=31
```

= associative array (hash)

# constructors IV.

---

```
m="jan"
```

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

```
...
```

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

... combination of indexed and hash array



# table library

---

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

# table library II.

---

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

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

# table – output

---

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

# array to hash

---

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

```
h = {}
```

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

result is a set

# array to 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
```

result is a multiset

# ascending, descending, or...?

---

```
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  
)
```

# functions II.

# function as a data type

---

```
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))
```



# function as a data type

---

```
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)
```

# functions: iterators and closures

---

- iterating function enable traversing through data (tables, files)
- two functions:
  - iterator (visible from the main scope)
  - internal function
- existing iterators:  
pairs, ipairs (see tables); lines (see files)
- own iterators

# own iterator

---

Let's create iterator returning data from only even indices:

```
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
```

files

# files and OS

---

- logical and physical point of view in files
- dependence on hardware (physical)
- independence on hardware (logical)
- file names
- file properties

# three criteria

---

- control characters (text/binary)
- handling (modes)
- access to data

# text and binary files

---

- according to the use of control characters:
  - text files
  - non-text files with the specified data type
  - non-text files with no type specified
- text files as character files
- internally organised into lines
- the end of the lines (OS)

# file handling

---

- read-only files
- write-only files
- both read and write files
  
- input v output



# file handling

---

- "r" read-only mode, default mode
- "w" write enabled mode; overwrites or creates a new file
- "a" append mode that opens an existing file or creates a new file for appending
- "r+" Read and write mode for an existing file
- "w+" All existing data is removed if file exists or new file is created with read write permissions
- "a+" Append mode with read mode enabled that opens an existing file or creates a new file

# file handling – example

---

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

"\*all" "\*a" reads the whole file

"\*line" "\*l" reads the next line

"\*num-  
ber" "\*n" reads a number (including leading  
whitespaces)

*num* reads a string, its length is determined by the  
num values

# access to data

---

- files processed sequentially  
typically text files
- files with direct access

# file exists?

---

- no special function
- solved by reading of zero characters:

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

# text files

---

- typical case
- lines: line iterator functions:  
io.lines(), f:lines()
- CSV processing

# text files – example

---

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

# modules

# terminology

---

- standard libraries
- user libraries
- isolated piece of code
- interface (global)
- internal structures (local)
- implementation of functions
- initialising operations



# example of own library

---

```
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!
```

# principles

---

- methods (functions) belongs to a hash
- the hash is returned
- other ways possible, too
  
- joining the module:

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

# advantages

---

- collecting related functions to one whole
- sharing the code
- easier composition of new projects
- implementation of abstract data types

# abstract data types

# background

---

- determine the data components used
- determine the operations and their properties
- abstract from the implementation method

# benefits

---

- ADT is determined by what we want/need in it
- ADT can be implemented in different ways without affecting its behaviour
- ADT is implemented using a suitable data structure (DS)

# ADT overview

---

- Stack
- Queue
- Non-repeating set (Set)
- Set with repetition (MultiSet)
- ...

# Queue (FIFO = First-in, First-out)

---

- access only to the element at the beginning (front, head)
- insertion only at the end of the queue (end, tail)



# Queue (diagram of signature)

---

- main data type
- related data types
- data flow arrows
  
- $\Rightarrow$  interface

# Typical operations

---

- constructor (init)
- modifiers (put, get)
- queries (size)
- predicates (empty)

# Queue (axiomatic description)

---

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

# Queue (implementation)

---

```
local Q = {}

Q.init = function (t) local q = {}
    for _, l in ipairs(t) do table.insert(q,l) end
    return q
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
```

# Stack (LIFO = Last-in, First-out)

---

- access only to the element at the top
- inserting only at the top (top)

# communication with OS

# files

---

- discussed before
- configuration of SW: see text files
- Linux: preprocessing via commands
- all methods belong to the module `os`

# environmental variables

---

- reading, but no modification possible
- preparing own copy of environmental variables

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

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



# executing commands

---

- knowledge of OS commands necessary
- function returns true or nil

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

# command line parameters

---

- enumerated from the left
- #0 = name of the running programme
- indexed array arg
- options together with parameters

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

# other

---

## ■ date+time

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

# applications

# LuaTeX

---

- a TeX-based computer typesetting system which started as a version of pdfTeX with a Lua scripting engine embedded.
- internals accessible via Lua

# Lua in ConT<sub>E</sub>Xt

---

- ConT<sub>E</sub>Xt: an extension of the basic T<sub>E</sub>X
- embedded Lua interpreted
- Lua enables more sophisticated operation which are more complicated using the T<sub>E</sub>X or ConT<sub>E</sub>Xt tools
- printing (function context) to the output stream (PDF)
- inspect implemented

# Lua in ConT<sub>E</sub>Xt – example

---

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

# games

---

- why Lua: compiler and interpreter can be simply embedded into any application. (Or only their part.)
- frameworks for 2D (eg. LÖVE, Pygame)
- frameworks for 3D (eg. Pyglet)
- relations to other libraries (eg. OpenGL)



# Programming language Lua

Tomáš Hála

Teaching Material



Funded by  
the European Union