LUA PROGRAMMING LANGUAGE

Davide Perticone
FLC assignments A.A. 2020/2021

Table of contents

- History of Lua
- Some uses of Lua
- Basics
- Variables
- Tables
- Functions
- Closures
- Coroutines
- Array
- Matrices
- Sets
- Linked lists
- Metamethods

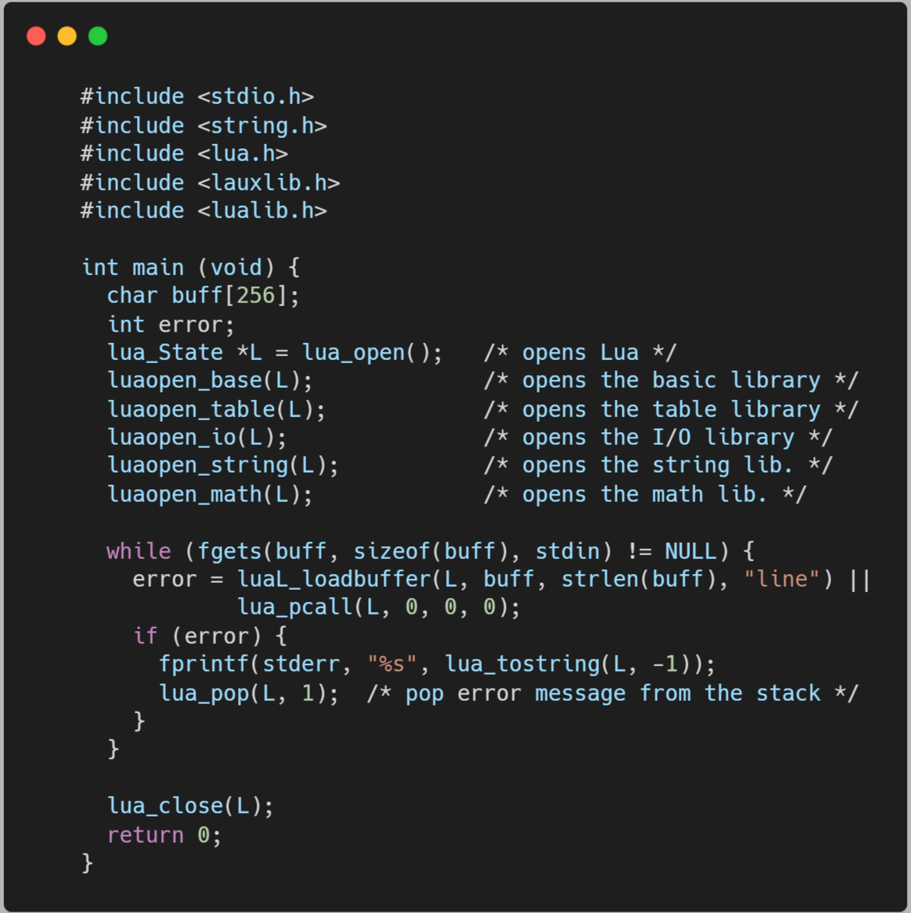
Running examples:

- LoopExample.lua
- FuncClose.lua
- Coroutines.lua
- Metatables.lua

HISTORY OF LUA

- Started as in-house project in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes.
- Lua is designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil.
- Designed from the beginning to be a software that can be integrated with the code written in C
- It does not try to do what C can already do but aims at offering what C is not good:
  - Good distance from hardware
  - Dynamic data structures
  - Ease of testing and debugging

Lua integration in C



```c
#include <stdio.h>
#include <string.h>
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int main (void) {
  char buff[256];
  int error;
  lua_State *L = lua_open();   /* opens Lua */
  luaopen_base(L);             /* opens the basic library */
  luaopen_table(L);            /* opens the table library */
  luaopen_io(L);               /* opens the I/O library */
  luaopen_string(L);           /* opens the string lib. */
  luaopen_math(L);             /* opens the math lib. */

  while (fgets(buff, sizeof(buff), stdin) != NULL) {
    error = luaL_loadbuffer(L, buff, strlen(buff), "line") ||
            lua_pcall(L, 0, 0, 0);
    if (error) {
      fprintf(stderr, "%s", lua_tostring(L, -1));
      lua_pop(L, 1);  /* pop error message from the stack */
    }
  }

  lua_close(L);
  return 0;
}
```

Some Uses of Lua:

- Game Programming (CryEngine, Angry Birds, World of Warcraft…)
- Scripting in Standalone Applications (Adobe’s Photoshop Lightroom)
- Scripting in Web
- Extensions and add-ons for databases like MySQL Proxy and
- MySQL WorkBench
- Security systems like Intrusion Detection System.

Info on Lua

What does it mean Lua? Is it a noun?
"Lua" (pronounced LOO-ah) means "Moon" in Portuguese.

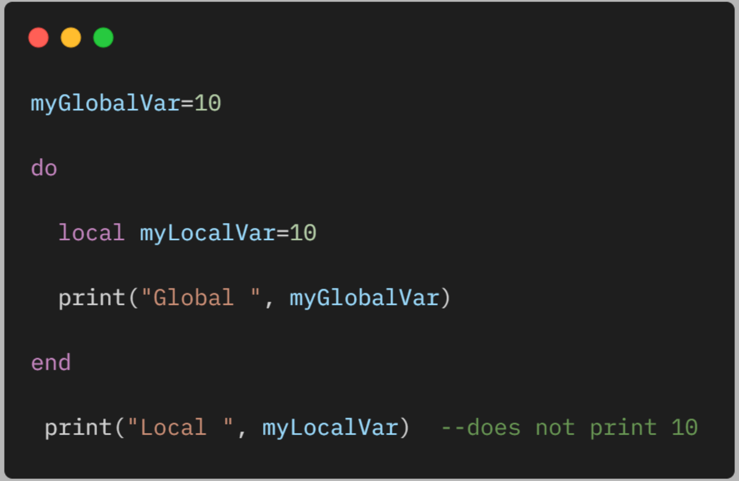
Is Lua fast?
Lua has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages.

Is Lua compiled or interpreted?
Lua programs are not interpreted directly from the textual Lua file, but are compiled into Lua bytecode, which is then run on the Lua virtual machine. The compilation process is typically invisible to the user and is performed during run-time. Then, Lua bytecode is interpreted. For the end-user, Lua can be considered as an interpreted language, but looking closely, it is compiled (and then interpreted).

Variables

- ***Global variables***: all variables are considered global unless explicitly declared as a local.

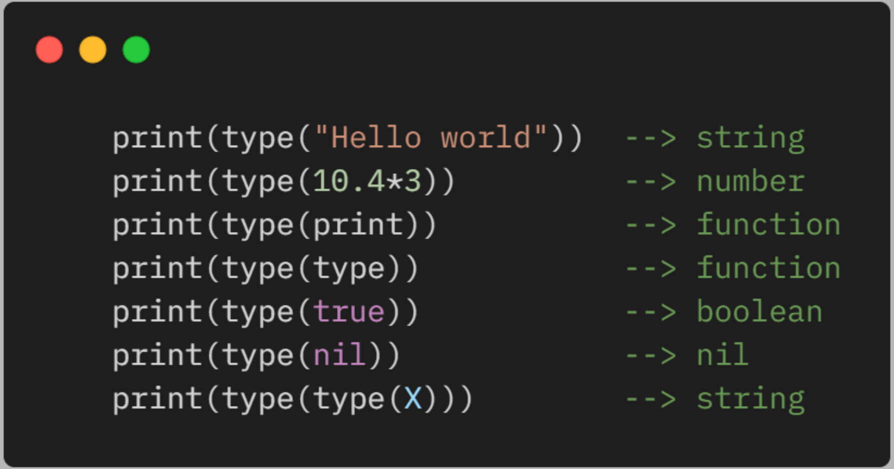
- ***Local variables***: when the type is specified as local for a variable then its scope is limited with the functions inside their scope.



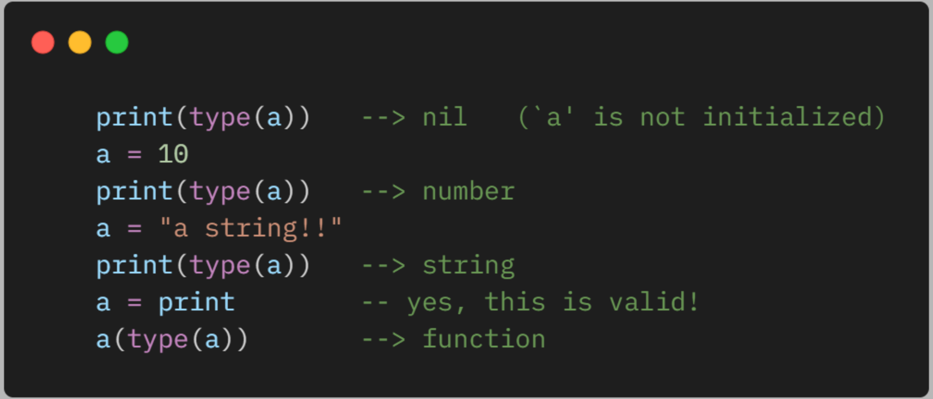
```lua
myGlobalVar=10

do

  local myLocalVar=10

  print("Global ", myGlobalVar)

end

 print("Local ", myLocalVar)  --does not print 10
```

Types and Values

- Lua is a dynamically typed language. There are no type definitions in the language; each value carries its own type.
- Eight basic types in Lua: *nil, boolean, number, string, userdata, function, thread, and table.*
- Variables have no predefined types; any variable may contain values of any type.



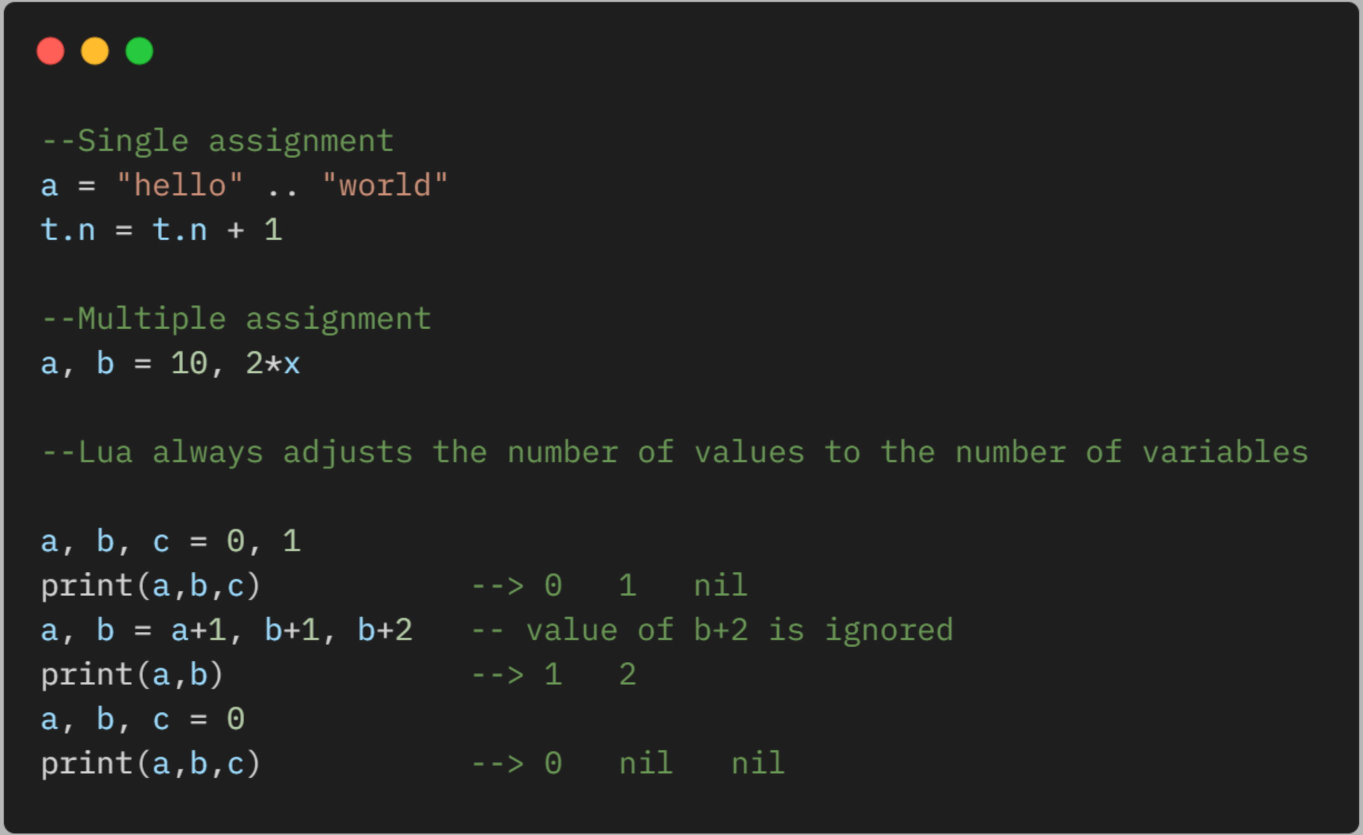
```lua
print(type("Hello world"))  --> string
print(type(10.4*3))         --> number
print(type(print))          --> function
print(type(type))           --> function
print(type(true))           --> boolean
print(type(nil))            --> nil
print(type(type(X)))        --> string
```



```lua
print(type(a))   --> nil   (`a' is not initialized)
a = 10
print(type(a))   --> number
a = "a string!!"
print(type(a))   --> string
a = print        -- yes, this is valid!
a(type(a))       --> function
```

Notice the last two lines: Functions are first-class values in Lua; so, we can manipulate them like any other value.

Assignment

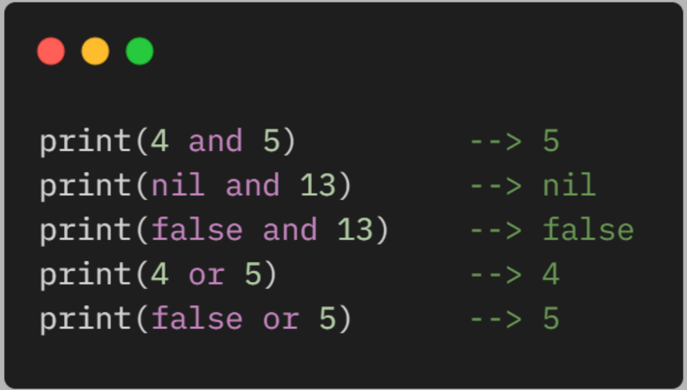


```lua
--Single assignment
a = "hello" .. "world"
t.n = t.n + 1

--Multiple assignment
a, b = 10, 2*x

--Lua always adjusts the number of values to the number of variables

a, b, c = 0, 1
print(a,b,c)           --> 0   1   nil
a, b = a+1, b+1, b+2   -- value of b+2 is ignored
print(a,b)             --> 1   2
a, b, c = 0
print(a,b,c)           --> 0   nil   nil
```

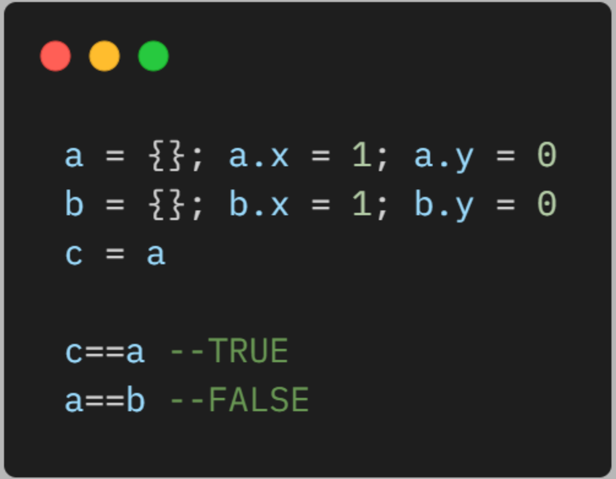
Expressions

- Arithmetic operations: \`+´ (addition), \`-´ (subtraction), \`*´ (multiplication), \`/´ (division), and the unary “-” (negation), ^ (just syntax to maintain simplicity, library gives meaning)

- Relational Operators: **< > <= >= == ~=** All these operators always result in true or false
- Lua compares tables, userdata, and functions **by reference**, that is, two such values are considered equal only if they are the very same object.
- The logical operators are ***and***, ***or***, and ***not***. Like control structures, all logical operators consider **false and nil as false and anything else as true**



```
print(4 and 5)          --> 5
print(nil and 13)       --> nil
print(false and 13)     --> false
print(4 or 5)           --> 4
print(false or 5)       --> 5
```

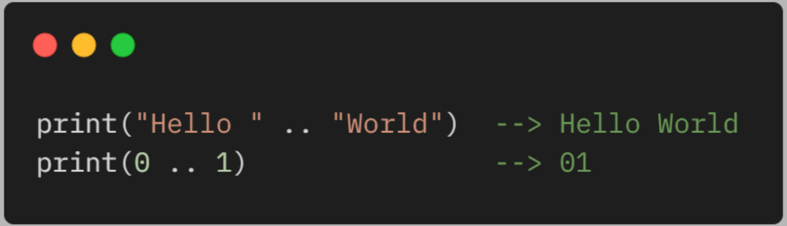


```
a = {}; a.x = 1; a.y = 0
b = {}; b.x = 1; b.y = 0
c = a

c==a --TRUE
a==b --FALSE
```

- The operator ***and*** returns its first argument if it is false; otherwise, it returns its second argument. The operator ***or*** returns its first argument if it is not false; otherwise, it returns its second argument

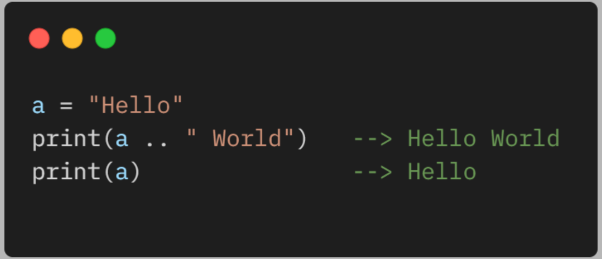
Expressions cont.

Lua denotes the string concatenation operator by ".." (two dots). If any of its operands is a number, Lua converts that number to a string.



```lua
print("Hello " .. "World")  --> Hello World
print(0 .. 1)               --> 01
```

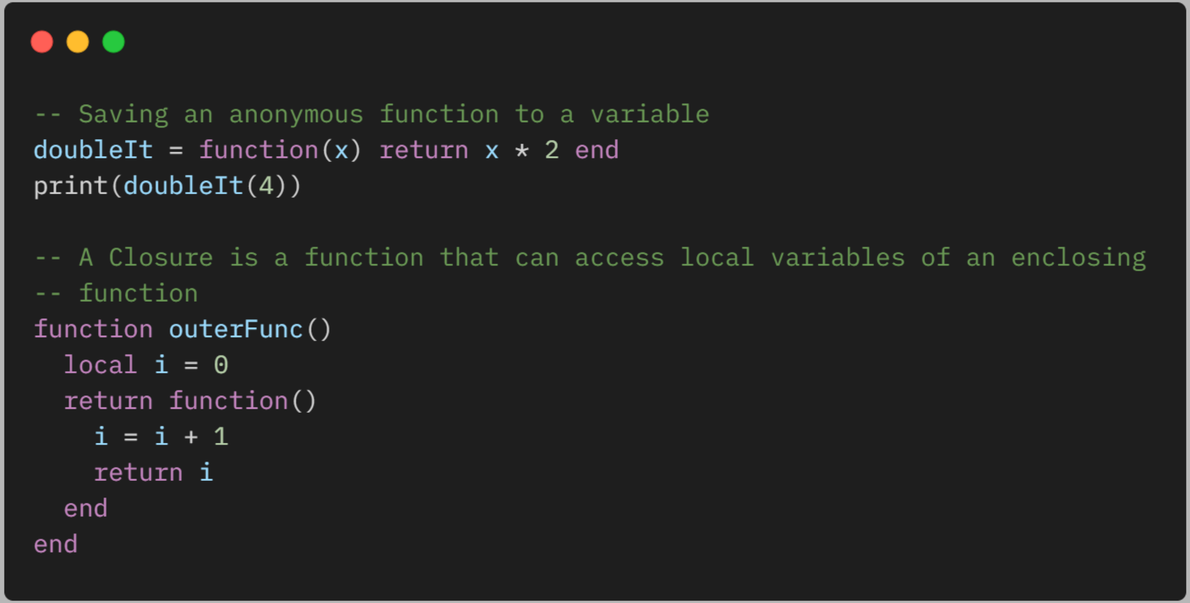
Strings in Lua are immutable values. The concatenation operator always creates a new string, without any modification to its operands:



```lua
a = "Hello"
print(a .. " World")   --> Hello World
print(a)               --> Hello
```

Functions

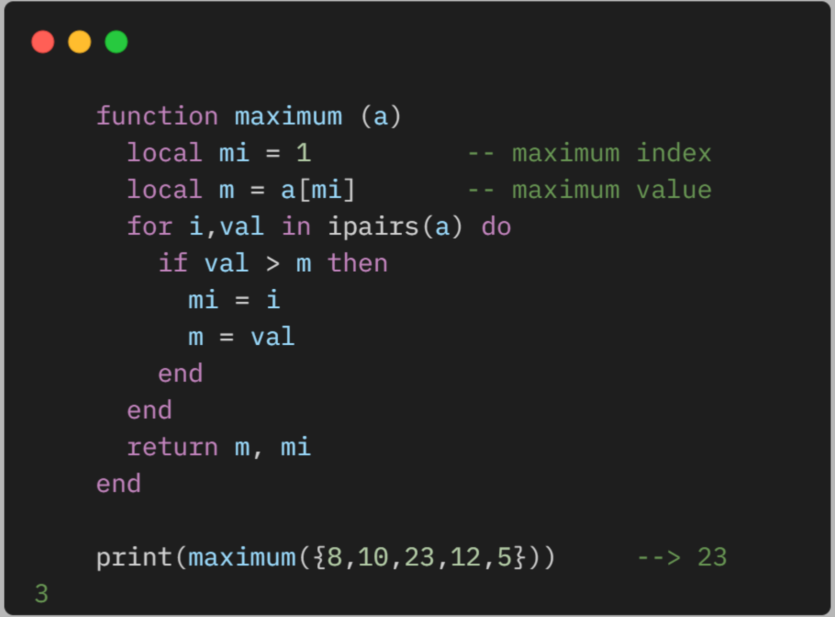
- Functions are first-class values
- Can be stored in variables
- Passed as arguments (to other function)
- Returned as results.
- Can be declared anywhere in the code



```lua
-- Saving an anonymous function to a variable
doubleIt = function(x) return x * 2 end
print(doubleIt(4))

-- A Closure is a function that can access local variables of an enclosing
-- function
function outerFunc()
  local i = 0
  return function()
    i = i + 1
    return i
  end
end
```

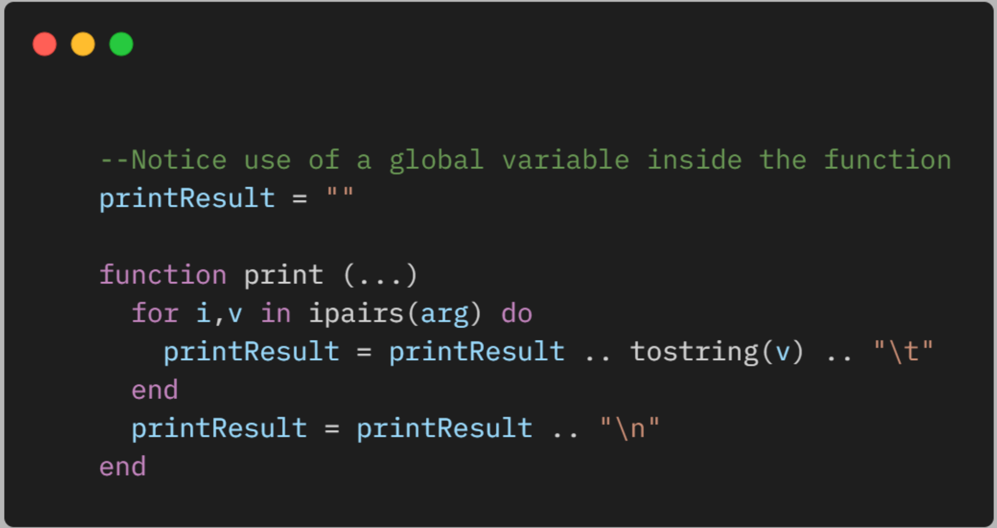
Multiple Results - Variable Number of arguments

- Functions written in Lua also can return multiple results, by listing them all after the return keyword.



```
function maximum (a)
  local mi = 1            -- maximum index
  local m = a[mi]         -- maximum value
  for i,val in ipairs(a) do
    if val > m then
      mi = i
      m = val
    end
  end
  return m, mi
end

print(maximum({8,10,23,12,5}))      --> 23
3
```

- Three dots (...) in parameter list indicate variable number or arguments
- When function called, parameters are collected and access through hidden table "arg",



```
--Notice use of a global variable inside the function
printResult = ""

function print (...)
  for i,v in ipairs(arg) do
    printResult = printResult .. tostring(v) .. "\t"
  end
  printResult = printResult .. "\n"
end
```

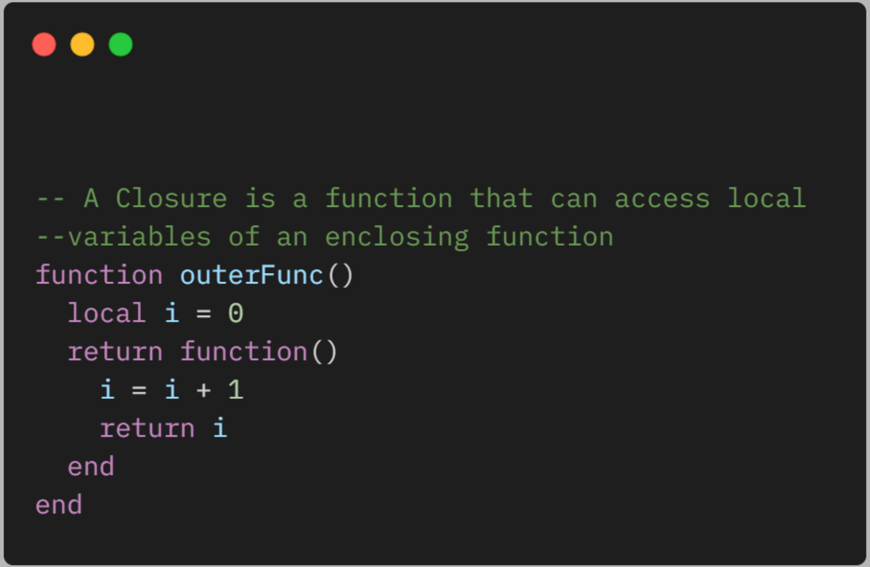
About Functions

- *Functions are first class values:*
  - Same rights as conventional values (strings, numbers).
  - Can be stored in (global and local) variables and tables.
  - Can be passed as arguments and retuned by other functions.
- *They have proper lexical scoping*
  - A function can access variables of its enclosing function (see closures, next slide).

Closures

- Lexical scoping allows enclosing function to access parent function.
- This features allows to declare closures:
    - Constructs where an inner function (returned as result) can access the parent local variables even after parent function has finished execution

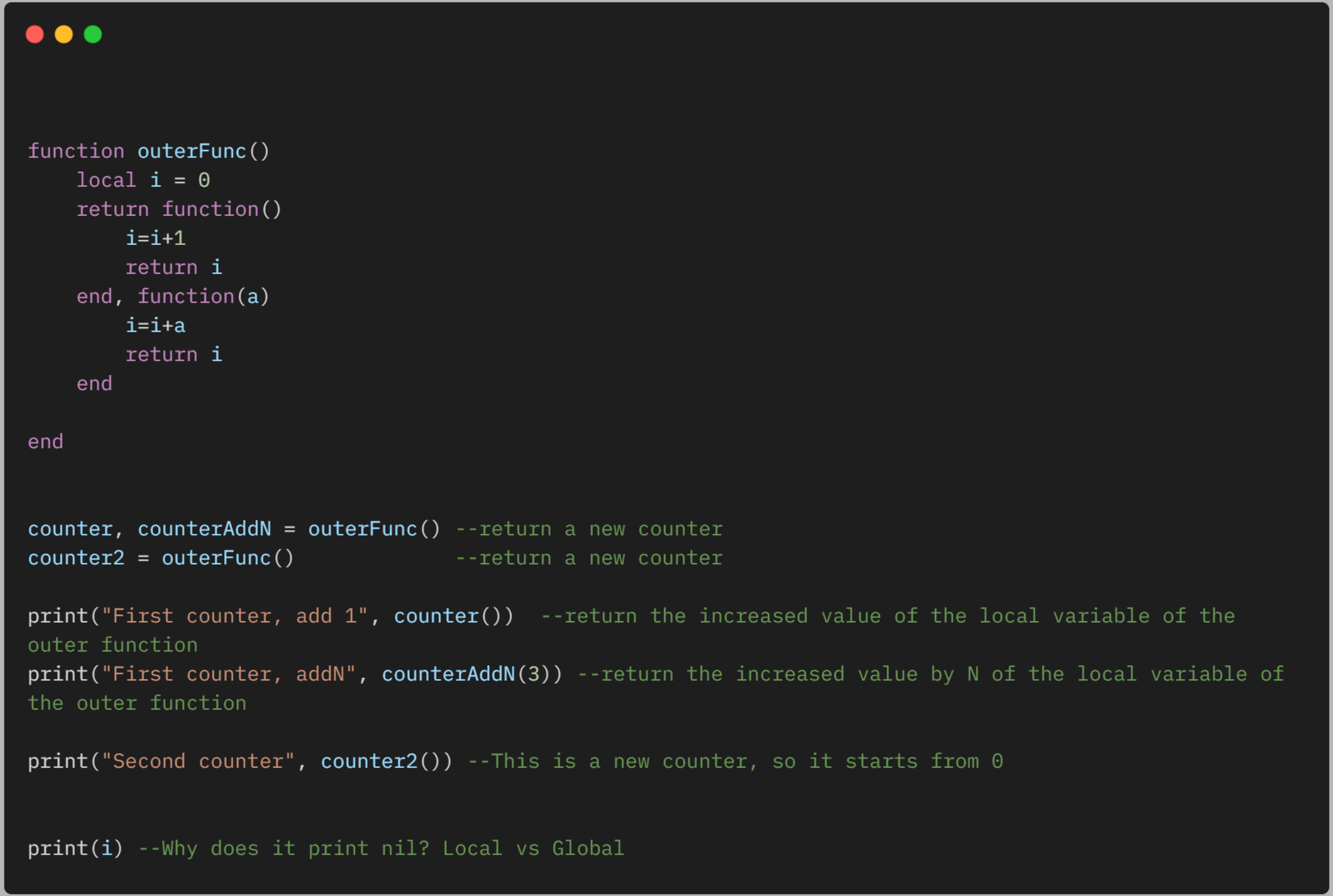
A closure construction typically involves two functions: the **closure itself**; and a **factory**, the function that creates the closure



```lua
-- A Closure is a function that can access local
--variables of an enclosing function
function outerFunc()
  local i = 0
  return function()
    i = i + 1
    return i
  end
end
```

Example next slide

Running example FuncClos.lua



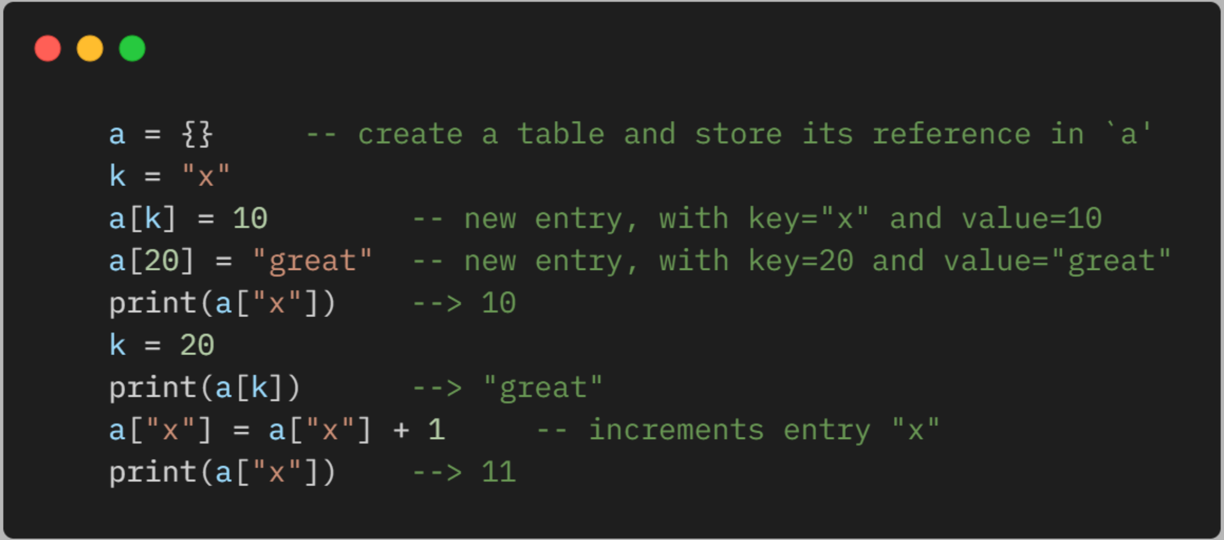
```lua
function outerFunc()
    local i = 0
    return function()
        i=i+1
        return i
    end, function(a)
        i=i+a
        return i
    end

end


counter, counterAddN = outerFunc() --return a new counter
counter2 = outerFunc()             --return a new counter

print("First counter, add 1", counter())  --return the increased value of the local variable of the
outer function
print("First counter, addN", counterAddN(3)) --return the increased value by N of the local variable of
the outer function

print("Second counter", counter2()) --This is a new counter, so it starts from 0


print(i) --Why does it print nil? Local vs Global
```

Tables

The table type implements associative arrays. An associative array is an array that can be indexed not only with numbers, but also with strings or any other value of the language, except nil.

Tables are the main (in fact, the only) data structuring mechanism in Lua, and a powerful one.
They are used to:

- Represent ordinary arrays
- Symbol tables
- Sets
- Records
- Queues
- Many other data structures.

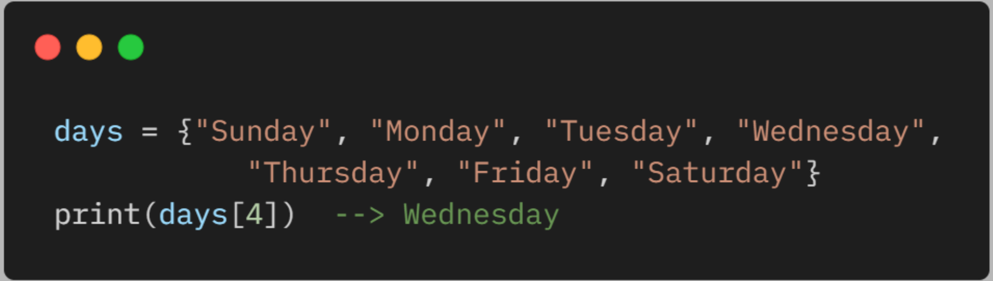


```lua
a = {}     -- create a table and store its reference in `a'
k = "x"
a[k] = 10        -- new entry, with key="x" and value=10
a[20] = "great"  -- new entry, with key=20 and value="great"
print(a["x"])    --> 10
k = 20
print(a[k])      --> "great"
a["x"] = a["x"] + 1     -- increments entry "x"
print(a["x"])    --> 11
```

Tables in Lua are neither values nor variables; they are objects. Same concept of Java

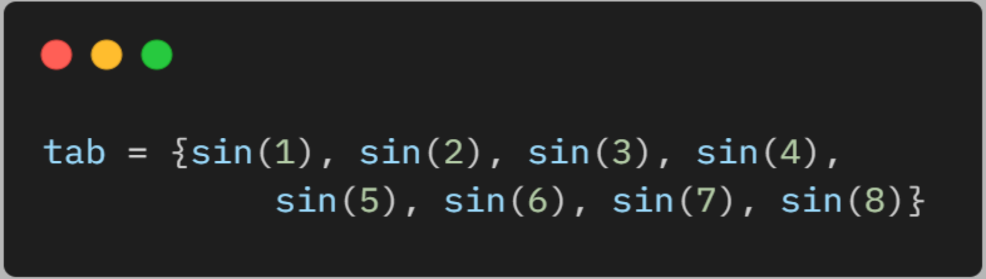
Constructors

Constructors are expressions that create and initialize tables. They are a distinctive feature of Lua and one of its most useful and versatile mechanisms. The constructor initializes days[1] with “Sunday”



```lua
days = {"Sunday", "Monday", "Tuesday", "Wednesday",
           "Thursday", "Friday", "Saturday"}
print(days[4])  --> Wednesday
```

Constructors do not need to use only constant expressions.



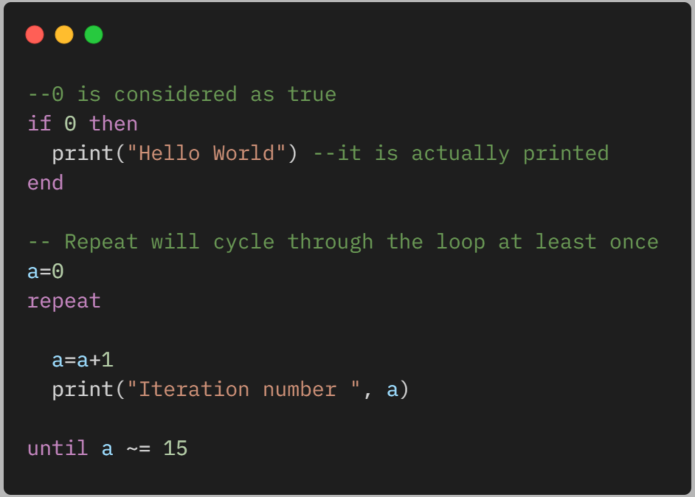
```lua
tab = {sin(1), sin(2), sin(3), sin(4),
         sin(5), sin(6), sin(7), sin(8)}
```

Control Structures

- If else elsif – conditional structures
- while, repeat until, for – iteration structures

All control structures have ***an explicit terminator: end*** terminates the if, for and while structures; ***until*** terminates the repeat structure.

The condition expression of a control structure may result in any value**. Lua treats as true all values different from false and nil.**

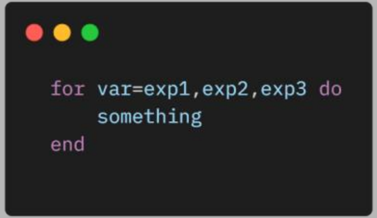


```lua
--0 is considered as true
if 0 then
  print("Hello World") --it is actually printed
end

-- Repeat will cycle through the loop at least once
a=0
repeat

  a=a+1
  print("Iteration number ", a)

until a ~= 15
```

Numeric for – Generic for

The for statement has two variants: the numeric for and the generic for.

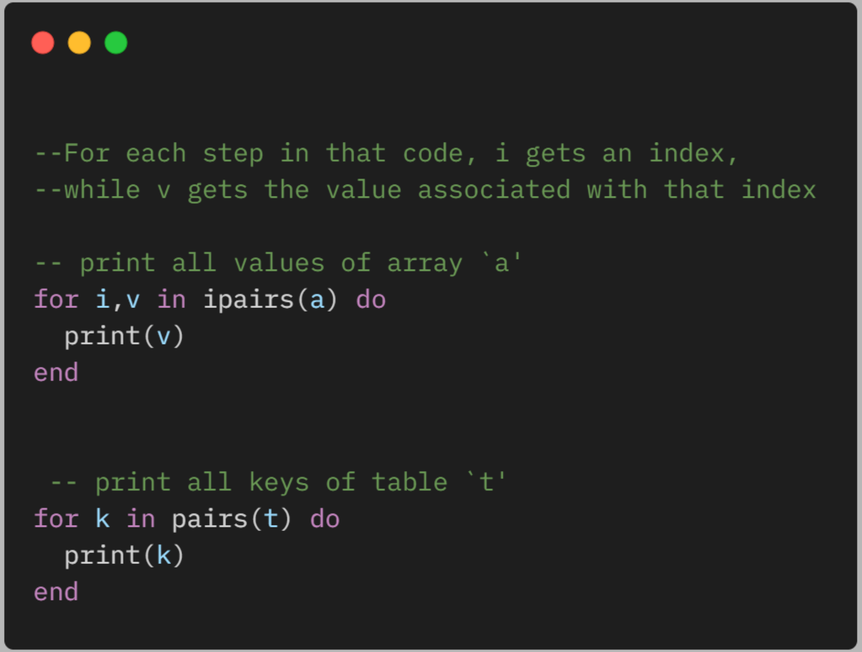
A numeric for has the following syntax:



```lua
for var=exp1,exp2,exp3 do
    something
end
```

That loop will execute something for each value of var from exp1 to exp2, using exp3 as the step to increment var. This third expression is optional; when absent, Lua assumes one as the step value.

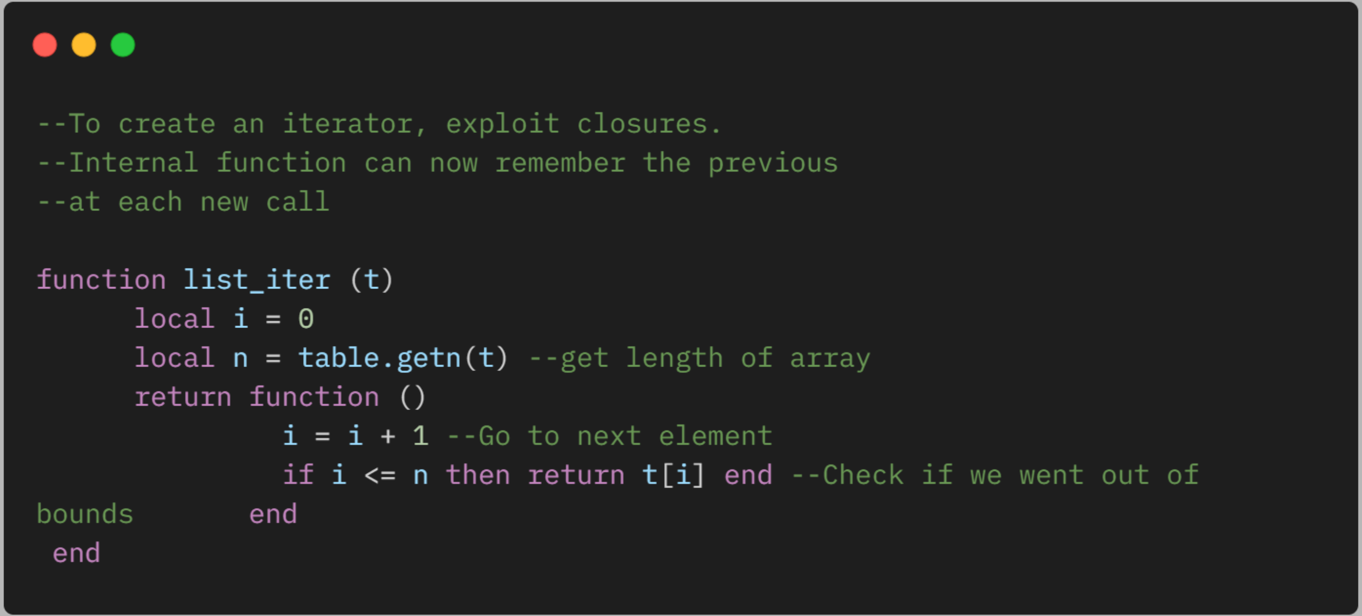
The generic for loop allows you to traverse all values returned by an iterator function.



```lua
--For each step in that code, i gets an index,
--while v gets the value associated with that index

-- print all values of array `a'
for i,v in ipairs(a) do
  print(v)
end


 -- print all keys of table `t'
for k in pairs(t) do
  print(k)
end
```

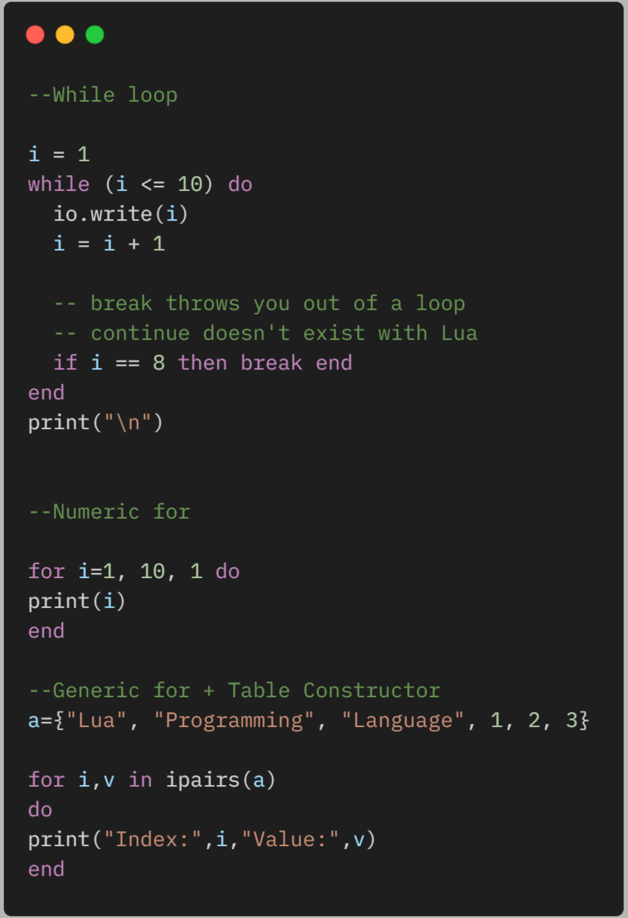
Iterators

An iterator is any construction that allows you to iterate over the elements of a collection. Lua represents iterators by functions: Each time we call that function, it returns a "next" element from the collection.

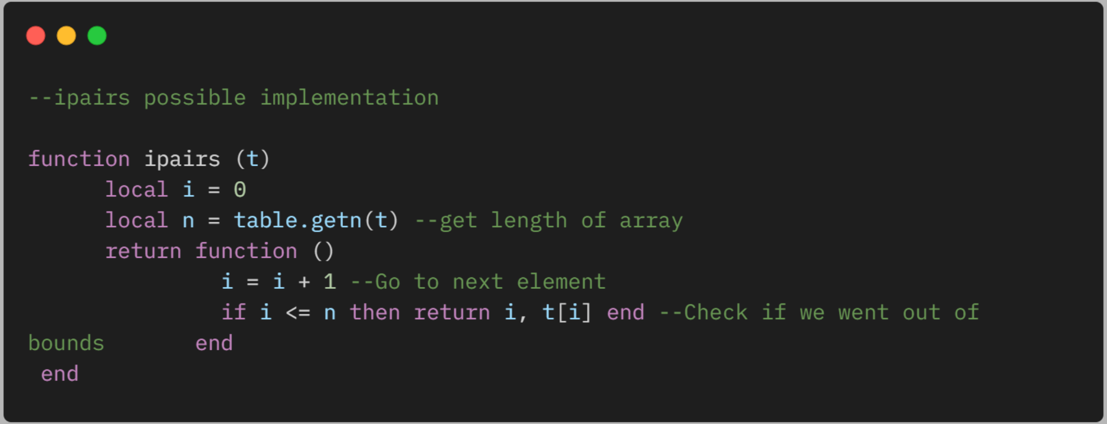


```lua
--To create an iterator, exploit closures.
--Internal function can now remember the previous
--at each new call

function list_iter (t)
      local i = 0
      local n = table.getn(t) --get length of array
      return function ()
               i = i + 1 --Go to next element
               if i <= n then return t[i] end --Check if we went out of
bounds       end
 end
```

Running example LoopExample.lua



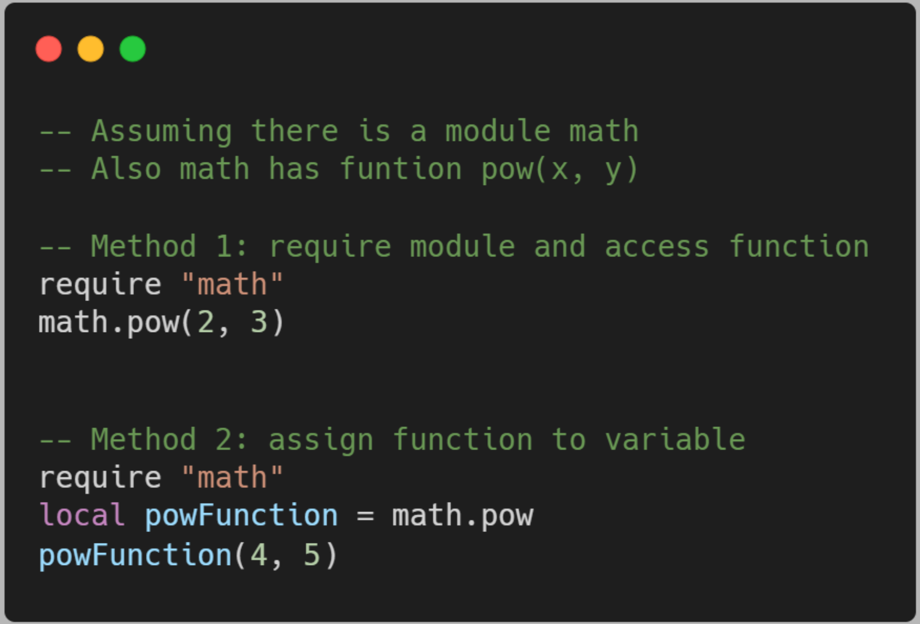
```lua
--While loop

i = 1
while (i <= 10) do
  io.write(i)
  i = i + 1

  -- break throws you out of a loop
  -- continue doesn't exist with Lua
  if i == 8 then break end
end
print("\n")


--Numeric for

for i=1, 10, 1 do
print(i)
end

--Generic for + Table Constructor
a={"Lua", "Programming", "Language", 1, 2, 3}

for i,v in ipairs(a)
do
print("Index:",i,"Value:",v)
end
```



```lua
--ipairs possible implementation

function ipairs (t)
      local i = 0
      local n = table.getn(t) --get length of array
      return function ()
               i = i + 1 --Go to next element
               if i <= n then return i, t[i] end --Check if we went out of
bounds       end
 end
```

The require Function

Lua offers a higher-level function to load and run libraries, called ***require***. First ***require*** searches for the file in a path; **require** controls whether a file has already been run to avoid duplicating the work.

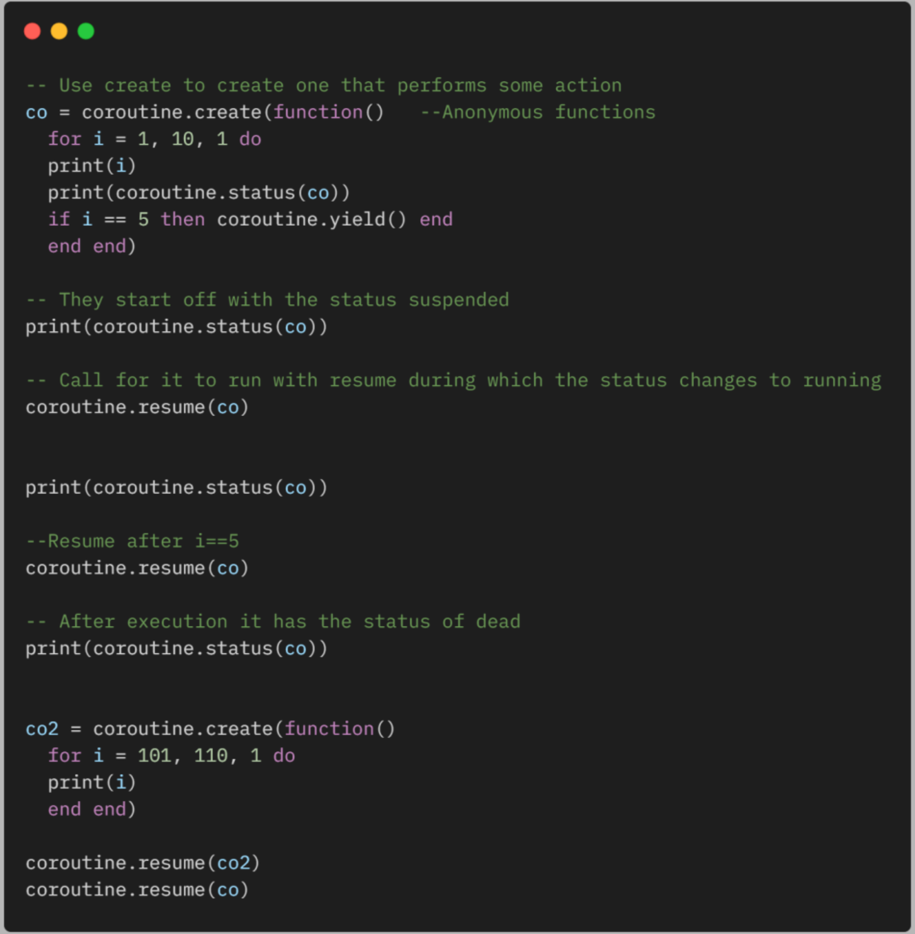


```lua
-- Assuming there is a module math
-- Also math has funtion pow(x, y)

-- Method 1: require module and access function
require "math"
math.pow(2, 3)


-- Method 2: assign function to variable
require "math"
local powFunction = math.pow
powFunction(4, 5)
```

Coroutines

- Lua offers all its coroutine functions packed in the ***coroutine*** table.
- The ***create*** function creates new coroutines. It has a single argument, a function with the code that the coroutine will run. It returns a value of type **thread**, which represents the **new coroutine**.
- Often, the argument to create is an ***anonymous function***.
- A coroutine can be in one of three different states: **suspended, running, and dead**. Upon creation, it starts in the suspended state.
- Coroutines are like threads except that they can't run in parallel.
- Coroutines are ***non-preemptive***. They cannot be stopped from outside.

Running Example Coroutines.lua



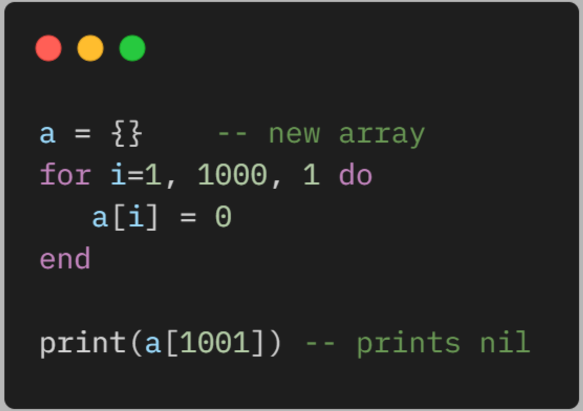
```lua
-- Use create to create one that performs some action
co = coroutine.create(function()   --Anonymous functions
  for i = 1, 10, 1 do
  print(i)
  print(coroutine.status(co))
  if i == 5 then coroutine.yield() end
  end end)

-- They start off with the status suspended
print(coroutine.status(co))

-- Call for it to run with resume during which the status changes to running
coroutine.resume(co)


print(coroutine.status(co))

--Resume after i==5
coroutine.resume(co)

-- After execution it has the status of dead
print(coroutine.status(co))


co2 = coroutine.create(function()
  for i = 101, 110, 1 do
  print(i)
  end end)

coroutine.resume(co2)
coroutine.resume(co)
```

Arrays

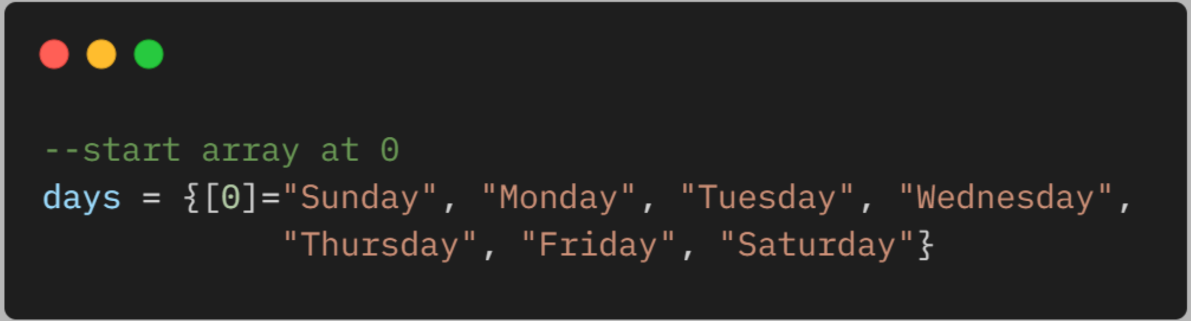
Lua implements arrays simply by indexing tables with integers.
Therefore, arrays **do not have a fixed size**, but grow as needed.
When initializing an array, the size is defined indirectly.

Arrays **start always from 1 instead of 0 if** not otherwise specified.

The Lua libraries adhere to this convention. It is possible to use library functions directly If arrays start from 1.



```lua
a = {}      -- new array
for i=1, 1000, 1 do
   a[i] = 0
end

print(a[1001]) -- prints nil
```



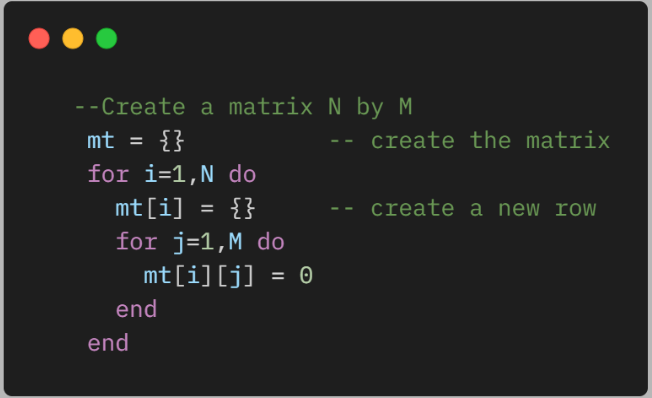
```lua
--start array at 0
days = {[0]="Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday"}
```

Matrices

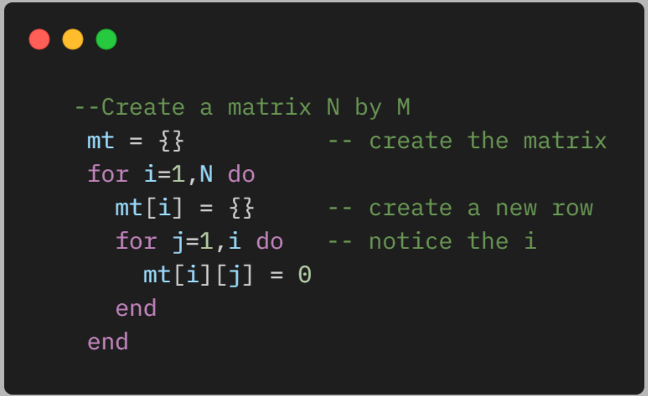
There are no direct methods to construct matrices in Lua. The easiest way is to use arrays of arrays, that is a table wherein each element is another table.

On the one hand, this is certainly more verbose than simply declaring a matrix, as done in C or Pascal. On the other hand, that gives more flexibility. For instance, it is possible to create a triangular matrix.

In this way, the triangular matrix saves half the space of a normal matrix.



```lua
--Create a matrix N by M
 mt = {}          -- create the matrix
 for i=1,N do
   mt[i] = {}     -- create a new row
   for j=1,M do
     mt[i][j] = 0
   end
 end
```

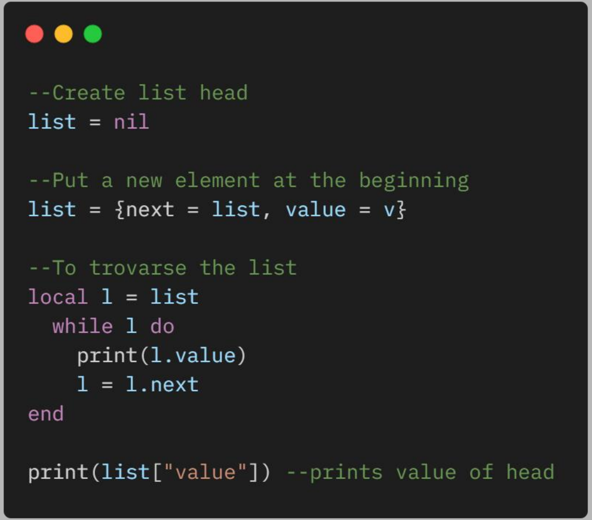


```lua
--Create a matrix N by M
 mt = {}          -- create the matrix
 for i=1,N do
   mt[i] = {}     -- create a new row
   for j=1,i do   -- notice the i
     mt[i][j] = 0
   end
 end
```

Linked Lists

Because tables are dynamic entities, it is easy to implement linked lists in Lua. Each node is represented by a table and links are simply table fields that contain references to other tables.

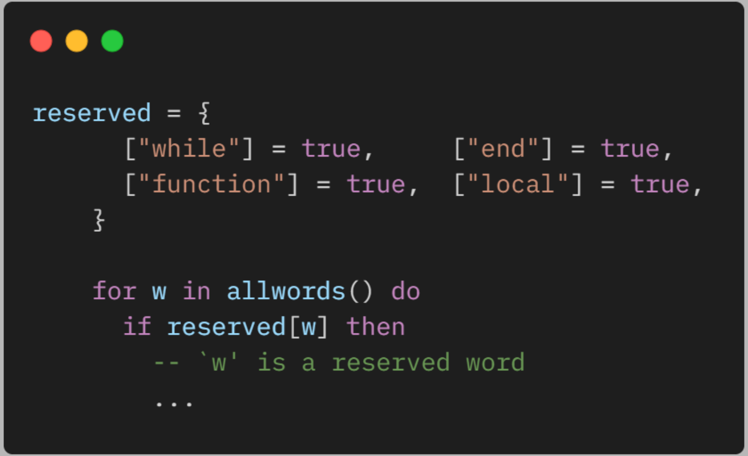
It is very easy to implement them, and the value can of course be another table, complex as needed.



```lua
--Create list head
list = nil

--Put a new element at the beginning
list = {next = list, value = v}

--To trovarse the list
local l = list
  while l do
    print(l.value)
    l = l.next
end

print(list["value"]) --prints value of head
```

Sets

- In Lua, an efficient and simple way to represent such sets is to put the set elements as indices in a table.
- The corresponding value is set to true
- Instead of searching the table for a given element, just index the table and test whether the result is nil or not.
- If nil, the element is not in the set, if true, it is.



```
reserved = {
      ["while"] = true,     ["end"] = true,
      ["function"] = true,  ["local"] = true,
    }

    for w in allwords() do
      if reserved[w] then
        -- `w' is a reserved word
        ...
```

Metamethods

A limited set of perations can be performed on tables:

- add key-value pairs
- check the value associated with a key
- Traverse all key-value pairs

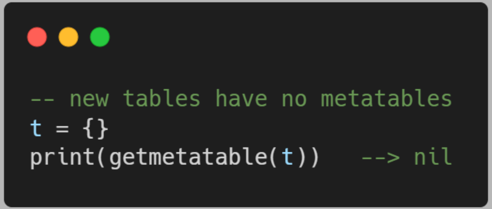
What about adding tables, compare tables, divide tables and so on?
It is not possible natively, unless…metatables are used.

Similar to overloading operators for classes in C++.

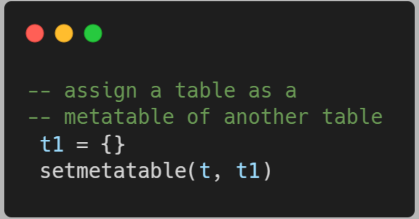
Metatables allow to change the behaviour of a table. If we want to add two tables following a specific logic, it is possible to define a metatable for one of the two tables (or both) containing a metamethod called __add.

If Lua finds __add when trying to add two tables, the corresponding metamethod is called to compute the sum. The order is decided based on the operands (if __add defined by first operand, use it, if not, use __add of second, if neither have __add Lua raises an error.

Each table in Lua may have its own metatable. By default, newly created table have no metatables.
Any table can be the metatable of any other table

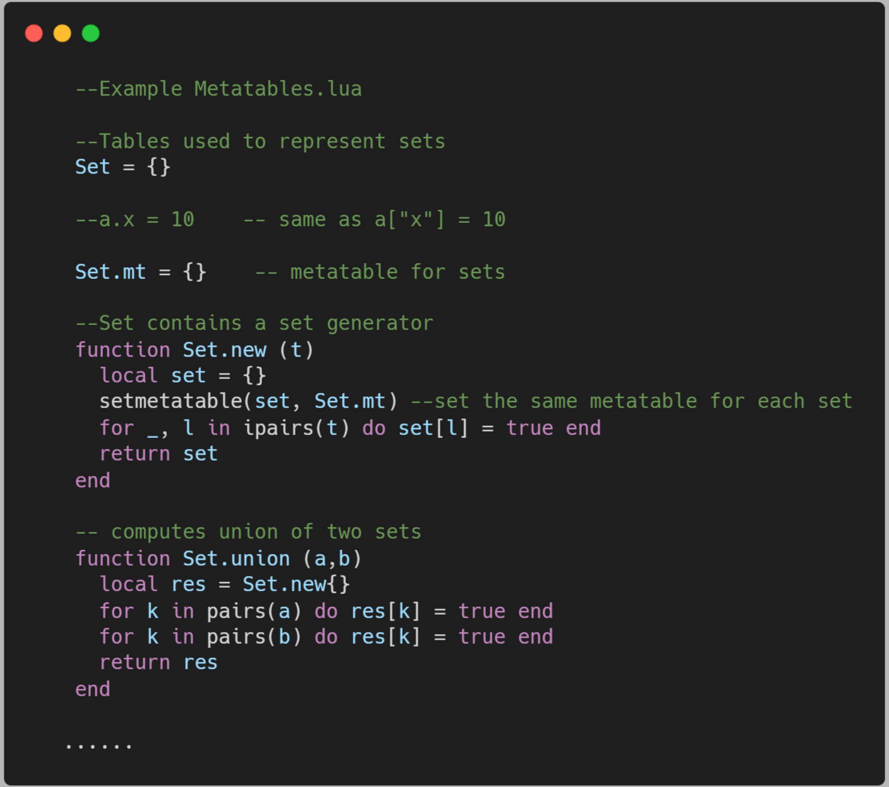


```lua
-- new tables have no metatables
t = {}
print(getmetatable(t))   --> nil
```



```lua
-- assign a table as a
-- metatable of another table
 t1 = {}
 setmetatable(t, t1)
```

Running example – Arithmetic metamethods



```lua
--Example Metatables.lua

--Tables used to represent sets
Set = {}

--a.x = 10    -- same as a["x"] = 10

Set.mt = {}    -- metatable for sets

--Set contains a set generator
function Set.new (t)
  local set = {}
  setmetatable(set, Set.mt) --set the same metatable for each set
  for _, l in ipairs(t) do set[l] = true end
  return set
end

-- computes union of two sets
function Set.union (a,b)
  local res = Set.new{}
  for k in pairs(a) do res[k] = true end
  for k in pairs(b) do res[k] = true end
  return res
end

......
```

What I implemented

Basic data type:

1. Number (Double, Integer)
2. nil (only in boolean expressions)
3. Table (1-D canonical array, fixed size)

Variables:

1. Global variables (not inside functions)
   1.1. Single and multiple definition and inline initialization
2. Local variables (only in functions, flow control instructions, local scopes)
   2.2. Single and multiple definition and inline initialization
3. Reassignment of global and local variables (within same type)
4. Array can be indexed by any expression

Operators:

1. Arithmetic
    1.1. Sum +
    1.2. Subtraction –
    1.3. Multiplication *
    1.4. Dision /
    1.5. Exponentiation ^ (of immediates through ^ operatore, with math.pow(x, y) for anything)
2. Logical
    2.1. AND and
    2.2. OR or
    2.3. NOT not (not implemented)
3. Relational
    3.1. Equality
    3.2. Inequality
    3.3. Less than
    3.4. Greater than
    3.5. Less than or equal
    3.6. Greater than or equal

Flow Control Instructions:

1. Loops:
   1. Numeric For loop
   2. While loop
   3. Repeat-until
   4. Nested loops (any depth)
2. Decision statement
   1. If then else (nested any depth)
   2. Nested if (any depth)
   3. Nested if then else (any depth)
   4. Nested if/if then else (any depth)

3. Loop Condition Supported: Single numbers, single variables, Boolean expressions (of any length), single array elements, mathematical expressions.

Functions:

1. Function declaration (anywhere in the code) (only NUMBER as parameters and return value)
2. Print function (Only string or only multiple numbers/variables)
3. Print(string.format()) (c-like printf).
4. “require” function implemented. It is possible to import file (libraries) and used them through library.namefunction(). Support also for global variables declared in libraries and error in case of multiple declarations.
5. Table constructor (accepts any arithmetic expression)

Instructions:

1. Declaration and inline initialization local and global variables
2. Assignment to local or global variables
3. return statement
4. function call
5. expression

Semantic error supported:

1. Variable not declared
2. Operation not supported by compiler(NOT)
3. Redeclaration of a variable into an array (limitation of compiler)
4. Array access to array not declared
5. Array access to variable (not array)
6. Redeclaration of function
7. Invocation of function not declared
8. Wrong number of parameters to function

Syntactical warning supported:

1. Error in if condition
2. Error in assignment of a variable
3. Missing } in array declaration
4. Missing) in function declaration
5. Missing return value of a function

Syntactical error supported

1. Expression between () not correct
2. Wrong variables list
3. Wrong loop variable initialization
4. Assign an array to a variable and vice versa
5. Declaration of global variables inside functions (Compiler limitation)
6. Print function without parameters
7. Pass an array to a function (Compiler limitation)
8. Duplication of require of a file

References

- Programming in Lua
  by Roberto Ierusalimschy
  Lua.org, December 2003
  ISBN 8590379817