Trabajo Práctico: Teoría de Lenguajes

Lenguaje: 

Integrantes:

- Costa Facundo - 99765
- Costa Gonzalo - 100243
- Marinaro Santiago - 97969

Corrector: Leandro Ferrigno

Fecha de Entrega: 18/06/2018

Origen

Lua es un lenguaje de programación interpretado, es de origen brasileño y nació en 1993.

Lua fue más allá de las expectativas de sus diseñadores. Ese éxito es debido a la simple decisión de implementación de mantener el lenguaje sencillo y pequeño, con una implementación fácil, rápida, portable y gratuita.

Lua fue diseñado por un pequeño comité de tres personas,Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, donde las nuevas características solo se implementan por unanimidad, o se dejaban para el futuro. Esto se hacía así para mantener el lenguaje lo más simple, de la simplicidad derivan otras cualidades - rápido, portable y de poco tamaño.

El motivo por el cual surge LUA

- **Nace DEL**

La primera experiencia fue en TeCGraf con un lenguaje proveniente de la aplicación de entrada de datos. Los ingenieros de PETROBRAS (la compañía de petróleo) necesitaban preparar archivos datos de entrada para hacer varias simulaciones al día. Este proceso era aburrido y propenso a errores, debido a que los programas de simulaciones tenían código heredado que necesitaban archivos de entrada con un formato estricto, típicamente números en columnas sin ninguna identificación que significa cada uno.

Petrobras le pidió a TeCGraf de crear una interfaz gráfica para este tipo de datos. Para simplificar el desarrollo de la interfaz decidieron hacer el código en una forma uniforme en lenguaje declarativo para poder describir cada tarea de entrada de datos, a esto lo llamaron DEL (data entry language).

- **Nace SOL**

Al mismo tiempo comenzaron a trabajar en otro proyecto llamado PGM un generador de informes. Como sugiere el nombre los reportes generados por

este programa eran altamente configurables, que las hacían el usuario final. El programa debía poder ser corrido como en MS-DOS y decidieron que la mejor manera de configurar esta aplicación era mediante un lenguaje especializado que lo llamaron SOL (simple object language).

Nace LUA

En Marzo de 1993, terminaron la primera implementación de SOL, pero no fue entregado. A mediado de ese año se dieron cuenta de que tanto DEL como SOL podían ser combinados en único lenguaje más poderoso. Se dieron cuenta que necesitaban un lenguaje de programación real con asignaciones, controlar asignaturas y subrutinas.

Como el potencial de usuarios no eran programadores profesionales, el lenguaje debería evadir sintaxis críptica. Finalmente el lenguaje debería ser portable. La portabilidad era una de sus principales fuertes, lo que pedía Petrobras no tenía un hardware específico, ya que tenían diferentes tipos de máquinas (PC-2, Windows 3.1, Macintosh, y Unix).

El nuevo lenguaje fue una versión modificada de SOL que uno de los desarrolladores decidió llamarlo LUA (luna en portugués).

Evolución

- **Lua v1.0**: Nace en **1993**, aunque nunca fue lanzado públicamente.

- **Lua v2.1: Febrero de 1995**. Incluye semántica extensible con retrocesos, una interfaz, y programación orientada a objetos. Los retrocesos son una función programada que Lua la llama cuando no sabe qué hacer.

- **Lua 2.2: Noviembre de 1995**. Incluyó long string, sintaxis extensible con funciones definidas, una interfaz de debug y el garbage collector.

- **Lua 3.0: 1997.** Inclusión de los tag methods, en reemplazo de los retrocesos, llamado durante la ejecución del programa para cambiar el comportamiento de Lua.

- **Lua 4.0: 2000.** Interfaz para crear un debugger y declaraciones múltiples parecido a Pascal o C, con asignación y control de estructuras y llamadas procedurales.

- **Lua 5.0: 2003.** Soporte multithreading , scoping y metatables en vez tag methods. Introducción de booleans, proper tail calls.

- **Lua 5.3: 2015.** Introducción de Enteros, Operaciones de Bits, librería UTF-8 y soporte 32 y 64 bits. Hasta antes de este momento se utilizaba Punto Flotante de Doble Precisión.

¿Para qué sirve LUA?

Lua es un lenguaje de programación extensible diseñado para una programación procedimental general con utilidades para la descripción de datos. También ofrece un buen soporte para la programación orientada a objetos, programación funcional y programación orientada a datos. Se pretende que Lua sea usado como un lenguaje de script potente y ligero para cualquier programa que lo necesite. Lua está implementado como una biblioteca escrita en C limpio (esto es, en el subconjunto común de ANSI C y C++).

Características

Sus fuertes se basan en:

- Extensibilidad: Fue diseñado para ser extensible mediante código LUA o código C externo.
- Simplicidad: Es simple de aprender, y pequeño, en linux llega a ocupar 220kb.
- Eficiencia: Tiene una implementación eficiente y es un lenguaje muy rápido.
- Portabilidad: El código fuente para correr en diferentes OS es el mismo, ya que se adhiere al estándar ISO (ANSI) C.

En principio, lo más destacable, es su tamaño, lo cual permite que sea el candidato perfecto para ser embebido dentro de una aplicación como puede ser un software de retocado de imágenes, ofimática o videojuegos.

Usos

Lua se puede dividir en tres grandes ramas:

- Lua embebido en una aplicación: Como en World of Warcraft, Nmap, Adobe Ligthroom, etc, utiliza la API de C.
- Lua estándar: Para procesamiento de textos, programas y proyectos, etc.
- Lua y C juntos: Utilizan Lua como una librería en C.

Características Básicas

Lua es un lenguaje interpretado por lo que no existe la compilación, ni se crea ningún ejecutable a partir de Lua, sino que necesitamos de un intérprete para leer las instrucciones.

Lua es un lenguaje de tipado dinámico y fuerte, por lo que no es necesario declarar el tipo de variable, lua asignará automáticamente el tipo de dato correcto.

Paradigmas que soporta

Lua es principalmente un lenguaje procedural, también es usado con diferentes paradigmas de programación, como: funcional, orientado a objetos, goal oriented, programación concurrente y data description.

- **Funcional**

Lua ofrece funciones de primera clase con lexical scoping. Por ejemplo:

```
(function (a,b) print(a+b) end)(10, 20)
```

Crea una función anónima que hace print de la suma de dos parámetros. El resultado va a ser la suma de 10+20 = 30. Todas las funciones en Lua son valores dinámicos anónimos, que se crean en tiempo de ejecución. También ofrece la manera convencional de crear funciones:

```
function fact (n)
         if n <= 1 then return 1
         else return n * fact(n - 1)
         end
end
```

A pesar de ser un lenguaje procedural, Lua usa funciones con frecuencia, muchas de ellas que se encuentran en la librería estándar y son high order.

- **Orientado a Objetos**

Lua solo tiene una estructura de datos, la tabla. Las tablas son de primera clase, arrays asociados creados dinamicamente. Dando a Lua un soporte parcial a los objetos. Para poder referenciar al objeto, lua utilizar una sintaxis sugar dedicada, el operador colon : , por lo que el receptor es pasado como un argumento extra como por ejemplo orb:foo() sería orb:foo(orb).

- **Goal Oriented**

Permite poder resolver un objetivo ya sea un objetivo primitivo ya sea objetivo primitivo o una disyunción de objetivos alternativos. Como por ejemplo: problemas de pattern matching.

- **Programación concurrente**

Lua evita los problemas tradicionales de multithreading, para ello utiliza Corrutinas. Una corrutina apilada es esencialmente un thread, es fácil de escribir un simple scheduler y con pocas líneas de código se puede completar el sistema.

- **Data Description**

Lua nació del lenguaje data-description llamado SOL, por lo que en Lua mantuvo la sintaxis de SOL original con pequeños cambios. Varias aplicaciones

usan Lua para data description, tambien algunos juegos frecuentemente lo usan para describir personajes y escenas.
Por ejemplo: HiQLab una herramienta para simular resonadores de alta frecuencia, usa Lua para describir mallas de elementos finitos.

Tablas

Las tablas en Lua son el único mecanismo de estructura de datos que existe, y es muy potente. Las tablas se utilizan para representar arrays, sets, records y otras estructuras de datos de manera simple, uniforme y eficiente.

- La tabla en Lua es esencialmente un array asociado, que acepta no solo números como índices sino también string y otros tipos de valores del lenguaje (a excepción de nil).

- Las tablas no son variables ni valores, son objetos. Se pueden ver como un objeto dinámico manejado por punteros.

Ejemplo

```
a = {} -- create a table and assign its reference
k = "x"
a[k] = 10 -- new entry, with key="x" and value=10
a[20] = "great" -- new entry, with key=20 and
value="great"
a["x"] --> 10
k = 20
a[k] --> "great"
a["x"] = a["x"] + 1 -- increments entry "x"
a["x"] --> 11
```

Inicialización

Las tablas en Lua se pueden inicializar de la siguiente manera:

```
a = {x = 10, y = 20}
```

```
--> o
a = {}; a.x = 10; a.y = 20
```

Table Library

También cuenta con una librería para el manejo de tablas

- table.insert (tabla, indice, valor) ó table.insert(tabla,valor)
  Nos permite insertar un valor en la tabla en el índice deseado.

- table.remove(tabla, indice)
  Nos permite remover un valor en el índice dentro de una tabla

- table.move(tabla,pos_ini,pos_final,pos_destino)
  Permite mover valores en una tabla desde una pos_ini a una pos_fin y moverla a una pos_destino (posición inicial).
  Tiene un parámetro adicional (una tabla) y permite copiarlo a una nueva tabla (el parámetro adicional)

- table.concat(table)
  Permite concatenar valores en una tabla, también admite parámetros adicionales.

- table.sort(tabla)
  Ordena la tabla basado en un comparador opcional, por defecto de menor a mayor

- table.pack() y table.unpack()
  Me permiten “empaquetar” y “desempaquetar” varias tablas, al hacerlo si hay algún nil en la tabla esta no se guarda

Estructuras de control

Lua ofrece un pequeño y convencional conjunto de estructuras de control en las cuales se encuentra: if, while, for y todas tienen una sintaxis con un terminador explícito: end.

- **if then else**

  if <condition> then <s1> else <s2> end
  if<condition> then<s1> elseif<condition>...
- **while**

  while <condition> do <s> end
- **repeat**

  repeat <s> until <condition>
- **for**

  for var = exp1, exp2, exp3 do <s> end
  for k,v in pairs (t) do <s(k,v)> end
- **goto**

Break y Return nos permite saltar fuera de un bloque. Goto nos permite saltar a casi cualquier punto de la función. el statement es:

goto <nombre de la función>.

Aunque existen algunas restricciones:

1. No podemos saltar a un bloque.
2. No podemos saltar fuera de una función.
3. No podemos saltar al scope de una variable local.

Tail Call

El tail cal es un goto vestido de call. Como los tail calls no usan espacio en el stack, el número de llamadas recursivas es ilimitado.

Podríamos hacer el siguiente codigo y nunca podría llenarse el stack:

```
function foo (n)
      if n > 0 then return foo(n - 1) end
end
```

Estructuras de datos

Las tablas en Lua no son una estructura de datos, pero son la estructura de datos, pudiendo representar cada estructura (arrays, listas, colas, etc) con tablas.

- **Listas**

En Lua podemos representar listas utilizando una tabla. Para acceder a cada valor usamos una key = numero y comienza desde 1. A la listas sin agujeros se lo llama secuencias.
También ofrece la función lenght haciendo #lista.

```
lista= {"uno","dos","tres" }

print(lista[1]) --> uno
print(lista[2]) --> dos
print(lista[3]) --> tres
print(#lista) --> 3
```

- **Arrays**

Para implementar un array convencional indexamos las tablas con integers, estas no tienen un size determinado e incrementan cuando lo necesita.

Ejemplo:

```
a={}
for i=-5,5 do
   a[i]=0
end
```

- **Lista enlazada**

Representamos cada nodo con una tabla. El enlazado es una simple tabla que contiene una referencia a otra tabla. Podemos crear una simple lista enlaza teniendo la referencia del siguiente y del valor:

```
--nodo
list=nil

--para insertar un elemento al comienzo de la lista
hacemos lo siguiente:
list = {next = list, value = v}

--para recorrer la lista
local l = list
     while l do
     visit l.value
     l = l.next
end
```

- **Matrices**

En lua hay dos maneras de representar una matriz, una es mediante un jagged array (array de arrays), que es una tabla en el que cada elemento contiene otra tabla.

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

La otra forma de representarlo es componiendo los dos índices en uno solo. Usualmente logramos hacer eso multiplicando el primer índice por una constante adecuada para luego agregarla al segundo índice.

```
local mt = {} -- create the matrix
for i = 1, N do
    local aux = (i - 1) * M
    for j = 1, M do
        mt[aux + j] = 0
    end
end
```

- **Colas y Cola Doblemente Terminada (Double-ended Queue)**

Colas:

Para implementar las colas, simplemente utilizamos las funciones insert y remove de la librería de tablas. Pero estas funciones se ordenan internamente y puede ser costoso para estructuras más largas.

Cola Doblemente Terminada:

Una manera más eficiente es crear dos índices, uno para el primer elemento y otro para el último. De esta manera podemos insertar y remover elementos en ambos extremos.

```
function listNew ()
    return {first = 0, last = -1}
end

function pushFirst (list, value)
    local first = list.first - 1
    list.first = first
    list[first] = value
end

function pushLast (list, value)
    local last = list.last + 1
    list.last = last
    list[last] = value
end
```

```
function popFirst (list)
    local first = list.first
    if first > list.last then error("list is empty") end
    local value = list[first]
    list[first] = nil -- to allow garbage collection
    list.first = first + 1
    return value
end

function popLast (list)
    local last = list.last
    if list.first > last then error("list is empty") end
    local value = list[last]
    list[last] = nil -- to allow garbage collection
    list.last = last - 1
    return value
end
```

Funciones

Son el principal mecanismo para abstraer statements y expresiones de Lua.

- Al llamarlas es necesario utilizar paréntesis ( ). A excepción de que reciban un solo parámetro y este sea un string o constructor de tabla.
- Pueden ser anónimas como no:
  function (...) <s> end ó function nombre (...) <s> end
- El pasaje de parámetros es por referencia.
- Son valores de primera clase: La función es como un valor con los mismos derechos que los valores más convencionales como los string o los números.

Ejemplo

```
print "Hello World" <--> print("Hello World")
```

```
dofile 'a.lua' <--> dofile ('a.lua')
print [[a multi-line <--> print([[a multi-line
message]] message]])
f{x=10, y=20} <--> f({x=10, y=20})
type{} <--> type({})
```

Multiple Retorno

Las funciones en Lua cuentan con retornos múltiples, es decir una función puede devolver más de un resultado.

Además al llamar a una función con múltiples retornos, si guardamos un solo valor, el resto se descarta.

```
function maximum (a)
          local mi = 1 -- index of the maximum value
          local m = a[mi] -- maximum value
          for i = 1, #a do
              if a[i] > m then
                  mi = i; m = a[i]
              end
          end
          return m, mi -- return the maximum and its
index
end

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

m = maximum({8,10,23,12,5})    --> 23     3-> discarted
```

Variadic Functions

Las funciones en Lua también pueden ser variadics, es decir puede tener una cantidad variada de argumentos. Esto se indica cuando escribimos en los parámetros de la función con tres puntos.

Ejemplo

```
function add (...)
        local s = 0
        for _, v in ipairs{...} do
                s = s + v
        end
        return s
end

print(add(3, 4, 10, 25, 12)) --> 54
```

Al hacer el for con {...}, esta tabla recolecta todo los parámetros pasados en la función.

Características Avanzadas

Pattern Matching

Lua no utiliza las expresiones regulares del estándar POSIX ni expresiones de Perl para Pattern Matching debido a la gran cantidad de líneas de código que tienen, que son más de la mitad de las librerías de Lua en código.

Pero sí ofrece una librería string basada en patrones muy poderosa

- string.find()

- string.match()
- string.gsub()
- string.gmatch()

string.find()

Devuelve las posiciones iniciales y finales de la palabra a buscar

string.find(string1,string2)

```
s = "hello world"
i, j = string.find(s, "hello")
print(i, j) --> 1 5
```

Cuenta con dos parámetros opcionales

El tercer parámetros es un índice que dice de donde comienza a buscar.
El cuarto es un booleano que indica si hace una búsqueda plana o utiliza los patrones.

```
string.find("a [word]", "[")
stdin:1: malformed pattern (missing ']')
string.find("a [word]", "[", 1, true) --> 3 3
```

En este ejemplo, el cuarto parámetro no está, por lo que intenta hacer una búsqueda con patrones, y no encuentra el patrón, caso siguiente se le pasa un booleano y hace la búsqueda plana de forma normal.

string.match()

Muy parecido a find(), solo que devuelve el string que estamos buscando

string.match(string1,string2)

```
print(string.match("hello world", "hello")) --> hello
```

Esto me puede servir por ejemplo si quiero sacar de un string solo una parte.
Por ejemplo:

```
date = "Today is 17/7/1990"
d = string.match(date, "%d+/%d+/%d+")
print(d) --> 17/7/1990
```

En este caso se utiliza el patrón “%d” para buscar decimales.

string.gsub()

gsub (global substitution) string.gsub(string,pattern,string2) es para buscar y reemplazar una palabra por otra.

```
s = string.gsub("Lua is cute", "cute", "great")
print(s) --> Lua is great
```

Además cuenta con un cuarto parámetro opcional que indica la cantidad de veces que vamos a reemplazar la palabra.

```
s = string.gsub("all lii", "l", "x", 2)
print(s) --> axx lii
```

Ejemplo con un patrón:

```
s = string.gsub("el dolar esta a 15 pesos", "%d%d", "28")
print(s)   --> el dolar esta a 28 pesos
```

string.gmatch()

gmatch (global match) devuelve una función para iterar todas las ocurrencias de un patrón sobre un string.

string.gmatch(string,”patron”)

```
string = "un ejemplo para mostrar"
palabras = {}
for w in string.gmatch(string, "%a+") do
    palabras[#palabras + 1] = w                    --
> {un,ejemplo,para,mostrar}
end
```

En este ejemplo, el gmatch devuelve un iterador para todas las ocurrencias de “%a+” es decir para una secuencia de letras (“palabra”), que luego dentro del for la guardo en la lista de palabras.

Patrones

Para Lua los patrones son string, que solo se van a comportar como patrones en funciones de Pattern Matching.

La lista de patrones:

.	all characters
%a	letters
%c	control characters
%d	digits
%g	printable characters except spaces
%l	lower-case letters
%p	punctuation characters
%s	space characters
%u	upper-case letters
%w	alphanumeric characters
%x	hexadecimal digits

+	1 or more repetitions
*	0 or more repetitions
-	0 or more lazy repetitions
?	optional (0 or 1 occurrence)

Caracteres especiales

Hay algunos caracteres mágicos que tiene significados especiales al usarlo en patrones.

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

Por ejemplo los corchetes [ ] nos permiten crear nuestro propios patrones, pudiendo hacer un patrón para números binarios '[10]'

Otro ejemplo es el ^ que sirve para buscar el complemento de cualquier caracteres que le pasaramos, por ejemplo '[^\n]' hace un matching con cualquier carácter que no sea un salto de línea.

Meta-tablas y Meta-métodos

Lua tiene una cantidad delimitada de operaciones. Podemos agregar números, concatenar strings, insertar key,value en tablas, etc. Pero no nos permite agregar tablas, comparar funciones y llamar a un string, a menos que usemos metatablas.

Meta-tablas:

Las metatablas nos permite cambiar el comportamiento de un valor cuando se enfrenta a una operación desconocida. Por ejemplo: usando metatablas podemos definir como Lua computa la expresión a + b, donde a y b son tablas.

```
mt = {}
tabla1 = {1,2,3}
tabla2 = {8,3,10}

setmetatable(tabla1,mt)
setmetatable(tabla2,mt)

function union (a, b)
     res={}
     aux={}
     for _,v in pairs (a) do aux[v]=true end
     for _,v in pairs (b) do aux[v]=true end

     for k,_ in pairs(aux) do
          table.insert(res,k)
     end
     return res
end

mt.__add = union  --seteo el metamethodo para la union

tabla3 = tabla1 + tabla2
```

```
for _,v in pairs(tabla3) do
    print(v)              --> un ejemplo para mostrar
end
```

Meta-métodos:

Es el metodo o funcion que se encarga de darle este comportamiento a las meta-tablas.

- **Aritméticos**

Son los métodos usuales que se usan para sumar, dividir, multiplicar, etc.
En los cuales se encuentran: __add (suma), __mul (multiplicación), __sub (resta), __div (división), __idiv (division de entero), __unm (negación), __mod (modulo), __pow (exponenciación), entre otros.

- **Relacional**

Son los operadores relacionales: __lt (less than), __le (less than / equal to), __eq (equal).

- **Librería**

Son los meta-métodos que se usan en Lua comúnmente:
__tostring (cuando haces print, llama a este método) y __pairs (cuando recorremos una tabla).

- **Table-access**

Lua también ofrece una manera de cambiar el comportamiento de las tablas para dos tipo de situaciones: el acceso y la modificación de campos ausentes en una tabla.

__index:

Al acceder a un campo que no existe en una tabla, nos devuelve Nil. Pero lo que pasa en realidad es que se fija que tiene un metamethod index, si existe, este nos devolvera el resultado.

```
t={}
print(t.x)  --> nil

-- Un ejemplo de herencia --

prototype = {x=0,y=0,ancho=100,largo=100}   --> Ejemplo de una ventana

mt={}
t2={x=10,y=10}
setmetatable(t2,mt)

mt.__index = function(_,key)
     return prototype[key]
end

--mt.__index = prototype    --> Es muy comun que lua hizo un shorcut para hacer esto mas simple

print(t2.ancho)   --> 100
```

__newindex:

Cuando asignamos un valor a un índice que no existe, ese lo asigna a la tabla. Pero en verdad lo que hace es llamar al metametodo newindex, y hace lo que dice newindex, que puede ser asignarlo a otra tabla.

```
t1 = {}
t1.x=10
print(t1.x)    --> 10
```

```
mt={}
t={x=10}
setmetatable(t,mt)

taux={}

mt.__newindex = taux --> Puede ser tambien la mt

t.y=50
print("t[y] tiene el valor",t.y)   --> t[y] tiene el
valor    nil
print("taux[y] tiene el valor",taux.y)  -->  taux[y]
tiene el valor50
```

ReadOnly

Con los meta-métodos newindex e index uno puede hacer una tabla que sea de solo lectura, asi no se podría modificar las tablas.

```
function readOnly(t)
     proxy = {}
     mt={}

     mt.__index = t   --> Pongo en index la tabla
original, para que cada vez que lea me de lo que tiene la
original
     mt.__newindex = function (t,k,v)  --> Cada vez que
quiero modificar algo en la tabla, salta error
          error("Intento de actualizar la tabla")
     end

     setmetatable(proxy,mt)
     return proxy
end
```

```
dias=readOnly{"Lunes","Martes","Miercoles","Jueves","Vier
nes","Sabado","Domingo"}

print(dias[1]) --> Lunes

ok,msg= pcall( function() dias[1]="otro dia" end )

print(msg)  --> Intento de actualizar la tabla
print(ok)   --> false (lanzo un error)
```

Programación orientada a objetos (POO)

Nuestro único Objeto en Lua es la tabla.

Y la tabla como los objetos pueden tener un estado y un identificador (self) que hace que se comporte diferente ante otro objeto (tabla).

Clases

Lua no tiene el concepto de clases: donde cada objeto es instancia de una clase.

En cambio, LUA puede emular clases como en un lenguaje basado en Prototypes (como en Javascript)

En estos lenguajes, los objetos no tienen clase, sino que tienen un prototipo donde el primer objeto puede ver cualquier operación que no conoce.

Ejemplo

```
Cuenta = {balance = 0}      --> NUESTRO PROTOTIPO
```

```
function Cuenta:new ()
     obj = {}
     self.__index = self         --> Decimos que nuestro
indice seamos nosotros (Cuenta.__index = Cuenta)
     setmetatable(obj, self)          --> Seteamos a self
(Cuenta) como metatable
     return obj
end

function Cuenta:depositar (valor)
     self.balance = self.balance + valor
end

function Cuenta:retirar (valor)
     if v > self.balance then error "saldo insuficiente"
end
     self.balance = self.balance - valor
end

cuenta1 = Cuenta:new()
cuenta1:depositar(1000)
print(cuenta1.balance)  --> 1000

cuenta2 = Cuenta:new()
print(cuenta2.balance)  --> 0
```

En el ejemplo, el protoype es Cuenta. La función Cuenta:new crea un objeto (que va a ser nuestra cuenta) y setea el índice a self (es decir Cuenta) y la metatable también va a ser la Cuenta.

Al hacer dos instancias de cuenta, estas se comportan de manera distinta, teniendo un único prototype Cuenta.

El **operador colon ‘ : ’** es syntaxis sugar, y sirve para ocultar el primer parámetro de la función, haciendo que sea por default el self.

Herencia

La herencia es fácil de implementar

Ejemplo

```
Cuenta = {balance = 0}

function Cuenta:new (obj)
     obj = obj or {}
     self.__index = self
     setmetatable(obj, self)
     return obj
end

function Cuenta:depositar (valor)
     self.balance = self.balance + valor
end

function Cuenta:retirar (valor)
     if v > self.balance then error "saldo insuficiente"
end
     self.balance = self.balance - valor
end



CuentaEspecial = Cuenta:new()

cuentaE = CuentaEspecial:new{limite=1000}   --> Cuenta
Especial hereda de Cuenta (el self hacen referencia a
CuentaEspecial

function CuentaEspecial:retirar(valor)
     if valor - self.balance >= self:getLimit() then
```

```
          error "No se puede retirar"
     end

     self.balance = self.balance - valor
end

function CuentaEspecial:getLimit ()
     return self.limite or 0
end


cuentaE:retirar(200)
print(cuentaE.balance) --> -200

ok,msg=pcall(function()cuentaE:retirar(1800)end)

→ pcall Catchea el error

print(ok,msg)  --> false No se puede retirar
```

Al hacer **CuentaEspecial = Cuenta:new()** estamos instanciando Cuenta heredando todas las funciones

La ”magia” ocurre en la siguiente línea de código:

cuentaE = CuentaEspecial:new{limite=1000}

Al hacer esto, cuentaEspecial además de heredar de cuenta, cuando hacemos self, nos estamos refiriendo a CuentaEspecial, y cuando creamos el new, lo va a ir a buscar a Cuenta, ya que CuentaEspecial no lo tiene implementado.

Además estamos poniendo un límite, esto nos permite redefinir las función para que lo utilicen.

Privacidad

La implementación estándar de objetos de Lua, no nos provee de privacidad.

En parte es, que esta implementación se realiza con Tablas

Por un lado, Lua nos dice que si quieres acceder a algo que es privado no lo hagas. Que lo marques con un _ si es privado y uno va a saber si alguien lo utiliza mal.

Pero por otro lado, uno de los objetivos de Lua es que sea flexible, ofreciendo al programador mecanismos para realizar diferentes cosas.

Aunque la mayoría de los programadores no utilizan frecuentemente esta implementación.

```
function nuevaCuenta (initialBalance)

     local self = {balance = initialBalance}

     local retirar = function (v)
          self.balance = self.balance - v
     end

     local depositar = function (v)
          self.balance = self.balance + v
     end

     local getBalance = function () return self.balance
end

     return {retirar= retirar, depositar=depositar,
obtenerBalance=getBalance}
end

cuenta = nuevaCuenta(1000)
cuenta.depositar(200)
```

```
print(cuenta.obtenerBalance())  --> 1200
```

En esta implementación, se utilizan dos tablas, en una se guarda el estado en una tabla, que no se puede acceder, el resto de funciones se implementan y se devuelven en una segunda tabla.

Entonces self = {balance = balanceInicial} no se puede acceder y se devuelven todos los métodos en la otra tabla {retirar= retirar, depositar=depositar, obtenerBalance=getBalance}

Corrutinas

Una corrutina es similar a un hilo, es una línea de ejecución con su propio stack, variables locales. La principal diferencia es que los hilos corren en paralelo, mientras que las corrutinas son colaborativas.

Todas las funciones relacionadas a las corrutinas están almacenados la tabla coroutine. La tabla posee los siguientes métodos:

- **coroutine.create (f):** Crea una nueva corrutina con la función f.
- **coroutine.resume (co [, val1, ...]):** Resume la corrutina y pasa parámetros si es que los necesita.
- **coroutine.running ():** Retorna la corrutina en ejecución, o nil si es llamado en el main thread.
- **coroutine.status (co):** Retorna el estatus de la corrutina.
- **coroutine.wrap (f):** Crea una corrutina, pero devuelve una función que cuando se la llama resume la corrutina.
- **coroutine.yield (...):** Suspende la corrutina. El parámetro que se le pasa actúa como un return para la función.

La creación de una corrutina se hace mediante la función *create* que recibe por parámetro la función que debe ejecutar. Devuelve un valor del tipo *“thread”*

```
co = coroutine.create(function () print("hi") end)
print(type(co)) --> thread
```

El estado de la corrutina se puede chequear mediante la función *coroutine.status*. Puede tener uno de los siguientes estados:

- Suspendido
- Corriendo
- Normal
- Muerto

La tabla de corrutinas posee una función *yield* que permite suspender una corrutina que se está ejecutando.

```
co = coroutine.create(function ()
    for i = 1, 10 do
        print("co", i)
        coroutine.yield()
    end
end)
```

Lo que hace la corrutina es imprimir un número y suspender su ejecución dentro de un for. Va a permanecer suspendido hasta que se le diga explicitamente.

Lua ofrece corrutinas asimétricas, significa que la función para detener y reiniciar la ejecución son distintas.

Compilation, Execution, and Errors

Aunque lua es un lenguaje interpretado siempre se pre-compila el codiga fuente antes de ejecutarlo.

Para la precompilación se usa el programa luac que viene con la distribución estándar.

```
$ lua -o prog.lc prog.lua
```

Así es como se el archivo precompilado de lua.

```
$ lua prog.lc
```

Luego el intérprete lo ejecuta.

El programa luac ofrece muchas interesantes opciones. En particular la opción -l lista los opcodes que el compilador genera.

El código precompilado, si bien no siempre es más ligero, es mucho más rápido que el código fuente.

Errors

Como Lua en general es usado de forma embebida no puede crashear y terminar su ejecución cuando un error sucede. Por eso ofrece formas de manejarlo.

Cualquier condición inesperada que ocurre lua levanta un error. Los errores pasan cuando el programa trata de sumar valores que no son números, llama valores que no son funciones, indexar valores que no son tablas. También se puede de forma explícita con la función *error*, de la siguiente forma:

```
print "enter a number:"
n = io.read("n")
if not n then error("invalid input") end
```

La función de error sujeta a una condición es tan común que existe la función *assert* que hace eso mismo.

```
print "enter a number"
n = assert(io.read("*n"), "invalid input")
```

pcall and xpcall

La función **pcall (f, arg1, ...)** llama a la función pedida de forma protegida. Si un pasa error en f, no tira un error. Simplemente retorna el estado del error.

Ejemplo de pcall:

```
function myfunction()
    n = n/nil
end

if pcall(myfunction) then
    print("Success")
else
    print("Failure")
end
```

El resultado de esta ejecución es Failure.

Con **xpcall (f, err)**, además de llamar a la función de forma protegida, si ocurre un error, llama a la función err (que es un error handler), y retorna un estado de error.

Ejemplo de xpcall:

```
function myfunction ()
    n = n/nil
end

function myerrorhandler ( err )
    print( "ERROR:", err)
end

status = xpcall( myfunction, myerrorhandler )
print( status )
```

En la ejecución se imprimirá ERROR: y el código de error.

Garbage Collector

Lua posee un manejo de memoria automático. El programa puede crear objetos, pero no hay función para borrarlos. Borra automáticamente los objetos que se convirtieron en basura, con el garbage collector.

El recolector es invisible al programador, pero se le puede ayudar, los principales mecanismos para hacerlo es:

- Weak tables
- finalizers
- collectgarbage

Weak tables

Mecanismo para decirle al recolector de que una referencia no debería prevenir la eliminación del objeto. Si el objeto sólo se encuentra en una tabla devil el recolector lo eliminará.

Para crear un tabla devil se usa el campo _mode de su metatable.

El siguiente es un ejemplo:

```
a = {}
mt = {__mode = "k"}
setmetatable(a, mt) -- now 'a' has weak keys
key = {} -- creates first key
a[key] = 1
key = {} -- creates second key
a[key] = 2
collectgarbage() -- forces a garbage collection cycle
for k, v in pairs(a) do print(v) end --> 2
```

El valor de campo es un string, “k” para las claves y “v” para los valores.

Finalizers

Es una función asociada a un objeto que se invoca cuando el objeto es recolectado.

Lua los implementa mediante el meta-método __gc, como en el siguiente ejemplo:

```
o = {x = "hi"}
setmetatable(o, {__gc = function (o) print(o.x) end})
o = nil
collectgarbage() --> hi
```

Collector garbage

Es una función que permite cierto grado de control sobre el recolector.
Se trata de muchas funciones en un. Su primer argumento es opcional, un string y especificaciones sobre qué hacer.

Las opciones para el primer argumento es:

- stop: Para el recolector.
- restart: Reanuda el recolector
- collect: Realiza un ciclo de recolección
- step: Realiza algo de recolección. Con un segundo argumento, data, especificando el trabajo
- count: Retorna la cantidad de KB usados por Lua en ese momento.
- setpause
- setstepmul

Los parámetros pause y setstepmul controlan el comportamiento del recolector. PAuse maneja cuánto hay que esperar hasta volver a empezar el ciclo. Stepmul cuánto trabajo hace el recolector para cada kb allocado.

C - API

Como se mencionó con anterioridad, Lua es un lenguaje embebible, es decir es una librería que puede asociarse a otras aplicaciones. Además los programas que utilizan Lua, pueden implementar funciones haciéndolo extensible.

La C - API es una serie de funciones o macros, constantes y tipos, que permiten interactuar a C con Lua. Permita utilizar a Lua como librería de C y viceversa.

Sigue el modus operandi de C, el cual es diferente al de Lua. La API se enfatiza en la flexibilidad y simplicidad, a veces al costo de facilidad de uso. Tareas comunes pueden requerir de muchas llamadas a la API, lo cual de control total sobre los detalles.

Ejemplo en C utilizando el "intérprete al desnudo" de Lua

```
#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 = luaL_newstate(); /* opens Lua */
    luaL_openlibs(L); /* opens the standard libraries */

    while (fgets(buff, sizeof(buff), stdin) != NULL) {
        error = luaL_loadstring(L, buff) || lua_pcall(L,
0, 0, 0);
        if (error) {
           fprintf(stderr, "%s\n", lua_tostring(L, -1));
           lua_pop(L, 1);
```

```
            /* pop error message from the stack */
        }
    }
    lua_close(L);
    return 0;
}
```

En este caso hay tanto código Lua como código C juntos. Todo lo que es de lua tiene el prefijo lua_.

La librería de Lua no define variables globales de C. Mantiene todo los estados en la estructura dinámica lua_State. Todas las funciones dentro de Lua reciben un puntero a esta estructura y un argumento.

La función luaL_newstate crea un nuevo estado de Lua, cuando se crea, este entorno no tiene ninguna función predefinida, ni siquiera print. Para mantener Lua con poco tamaño, todas las librerías estándar vienen en paquetes separador, para que no tengamos que utilizarlas si no las necesitamos. La función luaL_openlibs abre las librerías estándar.

Sobre la **API - C** hay mucha más información para desarrollar, la misma se puede encontrar en el libro **Programming in Lua, Fourth Edition - by Roberto Ierusalimschy** en la **Parte IV. The C API**.

Programación para videojuegos

EL lenguaje Lua es muy usado para hacer motores de juegos, debido a su simple estructura y sintaxis. El garbage collector también colabora a la decisión de usarlo, ya que es útil para los juegos que consumen mucha memoria.

Lua fue honrado con un premio en 2011 en la Game Developer Magazine como mejor herramienta de desarrollo para la categoría de videojuegos.
Lua se convirtió en un lenguaje de programación muy popular, tanto que ha logrado una gran masa de desarrolladores en la industria de los juegos.

Los siguientes motores usaron Lua:

- Corona SDK
- Gideros Mobile
- ShiVa3D
- Moai SDK
- LOVE
- CryEngine

Cada una de ellos está basado en Lua, y poseen una gran cantidad de API disponible.

Corona SDK

Corona SDK es un motor de juegos para celular, soporta iPhone, iPad, y Android. Existe una versión gratis más limitada que puede ser usada para juegos chicos.

Provee una gran cantidad de usos, incluido los siguientes:

- Physics and Collision handling APIs
- Web and Network APIs
- Game Network API
- Ads API
- Analytics API
- Database and File System APIs
- Crypto and Math APIs
- Audio and Media APIs

ShiVa3D

Es uno de los motores en 3D que provee un editor gráfico diseñado para crear aplicaciones. Soporta múltiples plataformas, que incluye Windows, Mac, Linux, iOS, Android.

Algunas de sus mejores características incluye:

- Standard plugins
- Mesh modification API
- IDE

- Built-in Terrain, Ocean and animation editor
- ODE physics engine support
- Full lightmap control
- Live preview for materials, particles, trails and HUDs
- Collada exchange format support

LOVE

Es un framework que se usa para hacer juegos en 2D. Es gratis y open source. Soporta Windows, Mac OS X y Linux.

Provee las siguientes características:

- Audio API
- File System API
- Keyboard and Joystick APIs
- Math API
- Window and Mouse APIs
- Physics API
- System and timer APIs

Comparativas

En esta sección se hará una comparación de Lua con otros lenguajes de programación.

Python Vs Lua

Python:

- Python está mejor equipado, con gran cantidad de librerías.
- Tiene soporte Unicode más extenso.
- Tiene mucha más documentación para principiantes.
- Es sensible a la tabulación
- Tiene un modelo para POO, con metaclases, herencia múltiple.

Lua:

- Lua es muchisimo mas liviano, y usa menos memoria.
- Intérprete más rápido (Lua vs Python) y compilador más rápidos (LuaJIT vs Pyico).
- No es sensible al espaciado/tabulado en blanco.
- No dicta un sistema particular OO, pero te deja crear tu propio sistema con metha-methods.

Se puede ver que en cuanto espacio y uso de memoria Lua es superior a Python. Pero este último está mejor equipado y soporta POO.

Go Vs Lua

Características de Go:

- Go es un lenguaje compilado.
- Tipado estático.
- Tiene punteros, pero no una aritmética.
- La librería es más grande, no se limita a ANSI C.
- Corre en sistemas unix y Windows.

Características de Lua:

- Lua es interpretado.
- Tipado dinámico.
- No tiene punteros.
- Lua ocupa menos de 1MB.
- Corre en cualquier lado que tenga compilador que soporte ANSI C

Características comunes:

- Retorno de múltiples variables.
- Strings son inmutables
- Usan Corrutinas
- No tiene try/catch

Las únicas diferencias importantes entre GO y Lua es la compilación, el tipado, y que GO está mejor equipado con una librería. Tienen muchas similitudes, como el return de múltiples variables, el uso de corrutinas, y la ausencias de statements del tipo try/catch.

Casos de Estudio

En está sección se presentarán distintos casos de estudio.

VLC

Lua se puede utilizar para hacer scripts para el reproductor VLC.
Se puede hacer las siguientes cosas:

- Playlist
- Art Fetchers
- Interfaces
- Extensiones
- Service Discovery

Todos los módulos de VLC se encuentran en el objeto *vlc*. Por ejemplo, si quieres usar la función *info* de *msg* se hace de la forma:

```
vlc.msg.info( "This is an info message and will be displayed in the console" )
```

Fuente: https://github.com/videolan/vlc/tree/master/share/lua

Angry Birds

Para hacer el juego se usó mucho el lenguaje Lua, la mitad del juego se hizo en lua básicamente.

El uso de Lua permitió hacer cambios de valores y agregar bloques de forma fácil.

Fuente: Presentación de la Game Design Expo 2011
https://youtu.be/XqTvKFzvv1Y?t=8m28s

World Of Warcraft

En el World Of Warcraft Lua fue utilizado para crear Addons (Mods) que modifican la interfaz del juego.

La actual framework de la interfaz de World of Warcraft está hecha en Lua.
Y el propio Cliente de Blizzard nos provee una API (WOW API) para utilizar una lista de funciones y facilidades hechas en LUA.

WOWBench programado en Lua, utiliza la API de WOW para crear un entorno para debuggear los Addons creados por los desarrolladores de manera offline.

Fuente: http://lua-users.org/wiki/WorldOfWarcraft

Lua LÖVE

Es un framework para el desarrollo de juegos en 2D. Es gratis, se puede usar para fines comerciales sin limitaciones. Se utilizó para proyectos comerciales, game jams.

Los siguientes son algunos de los juegos hechos con Love:

- Mr. Rescue
- Move or Die
- Oh my giraffe
- Metanet Hunter
- Warlock's Tower
- Mari0

Lua Love posee los siguientes módulos:

- love.audio
- love.data.
- love.event
- love.filesystem
- love.font.
- love.graphics
- love.image

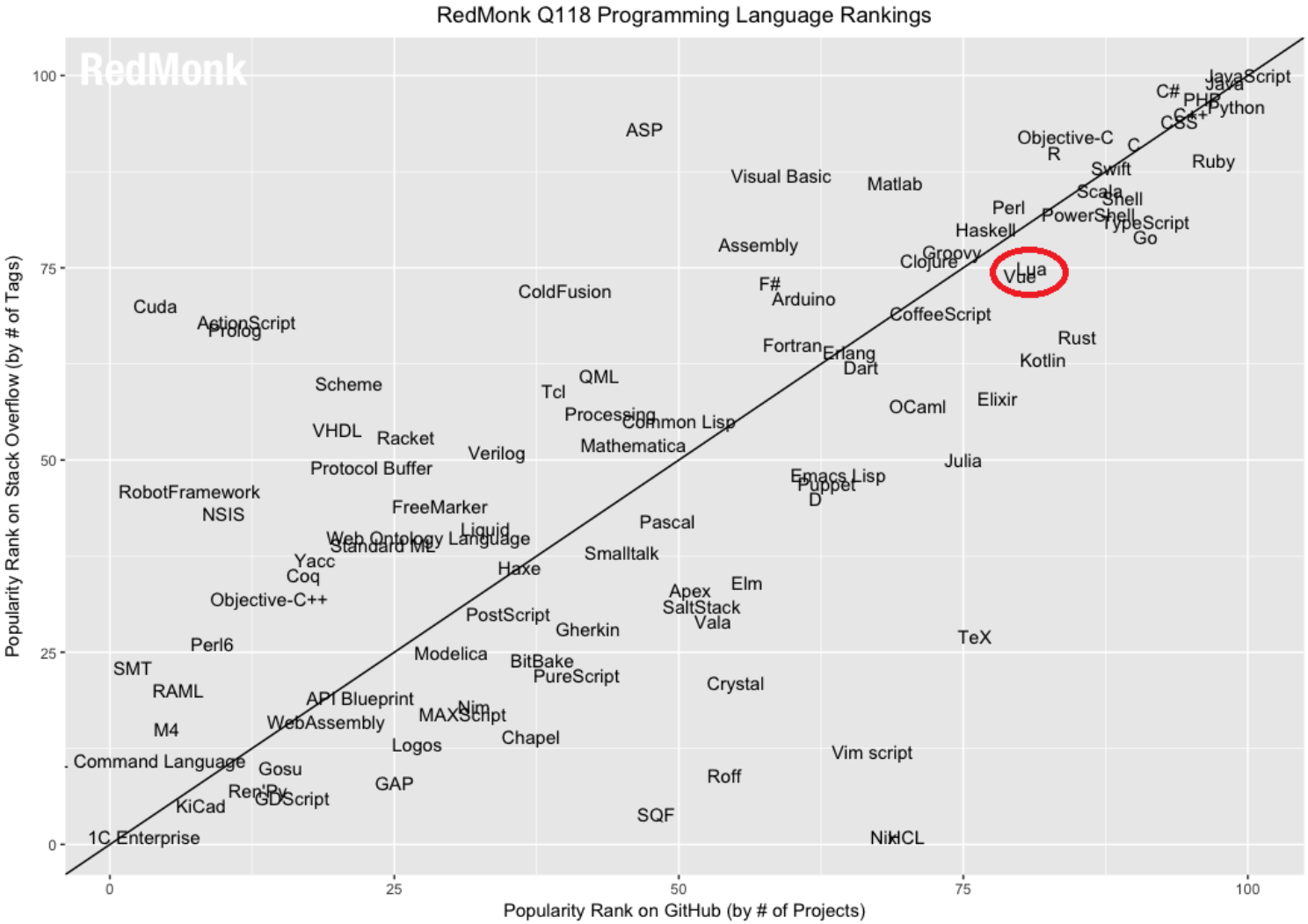
- love.joystick
- love.keyboard
- love.math
- love.mouse
- love.physics
- love.sound
- love.system
- love.thread
- love.timer
- love.touch
- love.video
- love.window

Fuente: https://love2d.org/

Estadísticas

Las siguientes tablas son estadísticas de popularidad de Lua:

Tabla de Popularidad - Números de proyectos en GitHub/ Trends en Stack



Oveflow

Se puede ver del gráfico:

1. JavaScript
2. Java
3. Python
4. PHP
5. C#
6. C++

7. CSS
8. Ruby
9. C
10. Swift
11. Objective-C
12. Shell
13. R
14. TypeScript
15. Scala
16. GO
17. PowerShell
18. Perl
19. Haskell
20. **Lua**

Fuente: https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/

Popularidad con qué frecuencia se busca tutoriales en Google.

12	↑↑↑	TypeScript	1.54 %	+0.4 %
13		VBA	1.37 %	-0.0 %
14		Scala	1.23 %	-0.1 %
15	↓↓↓	Visual Basic	1.22 %	-0.2 %
16	↑↑↑↑↑↑	Kotlin	0.93 %	+0.6 %
17		Go	0.92 %	+0.3 %
18	↓↓	Perl	0.76 %	-0.1 %
19	↓	Lua	0.43 %	-0.0 %
20	↑	Rust	0.37 %	+0.0 %
21	↓	Haskell	0.3 %	-0.1 %
22	↓↓↓	Delphi	0.27 %	-0.1 %
			© Pierre Carbonnelle, 2018	

Se puede ver que en ambas Tablas Lua se mantiene en los primeros 20 Lenguajes más populares.
Fuente: http://pypl.github.io/PYPL.html

Conclusiones

Las conclusiones que sacamos es que lua es un lenguaje con futuro. Dado las siguientes características:

- **Lua es un lenguaje robusto:**

Lua ha sido utilizado en muchas aplicaciones industriales (Adobe Photoshop Lightroom), con énfasis en sistemas embebidos (como Ginga middleware para la TV Digital en Brasil) y juegos (como World of Warcraft y Angry Birds, entre muchos otros).

- **Lua es rápido:**

Lua tiene una reputación de por su performance. El decir que es “tan rápido como Lua” es una aspiración de muchos lenguajes.

- **Es portable:** Pesa alrededor de 246K con las librerías estándares.

Lua corre en cualquier sistema que soporte Unix y Windows, también en teléfonos celulares (Android, iOS, BREW, Symbian, Windows Phone).

- **Tiene una sintaxis sencilla.**

- **Es un lenguaje embebible:** Es fácil de embebir en una aplicación. Posee una simple y bien documentada API que permite una integración con código escrito en otros lenguajes.

- **Se sigue trabajando en el lenguaje.**

Se sigue usando para el desarrollo de juegos, de los cuales algunos de ellos se pueden encontrar a la venta en Steam(Move or Die, Warlock’s Tower). Por eso pensamos que se seguirá usando en el futuro.

Fuentes

http://www.lua.org/
http://lua-users.org/
https://love2d.org/

Libro: Programming in Lua, Fourth Edition - by Roberto Ierusalimschy.

Casos de Estudio:

https://github.com/videolan/vlc/tree/master/share/lua
http://lua-users.org/wiki/WorldOfWarcraft
https://sites.google.com/site/marbux/home/where-lua-is-used

Estadísticas:
https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/
http://pypl.github.io/PYPL.html