TEORÍA DEL LENGUAJE

Informe Individual

Profesores responsables:

Ferrigno Leandro

Scarpinelli Ariel

Estudiantes:

Casal Romina (86429)

Contenidos

Introducción

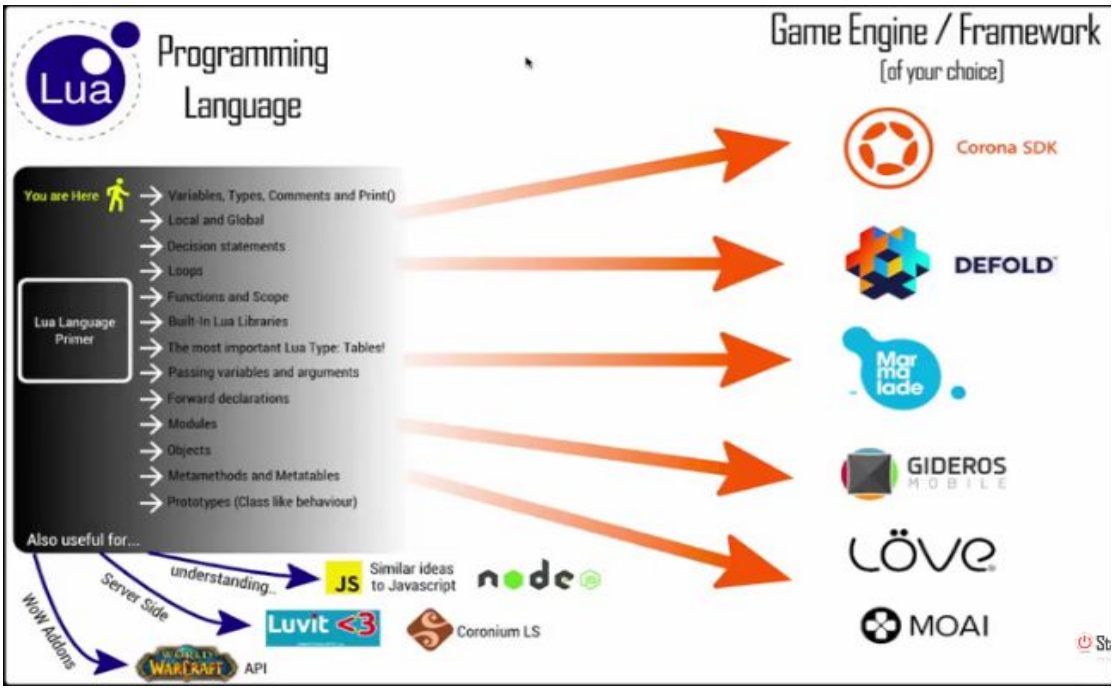
El presente informe es una recopilación de las características avanzadas del lenguaje Lua.

Se explican de manera general dos de sus principales usos, el desarrollo de servidores y juegos.

Como objetivo principal se busca explicar las características avanzadas para poder entender su aplicación en la resolución de los distintos problemas que se puedan presentar.

Por último se hace mención a las estadísticas recopiladas de distintas fuentes sobre el estado actual del lenguaje respecto a su posicionamiento en el ranking de lenguajes más usados, sus estadísticas respecto al mercado laboral y a su presencia en distintas comunidades y redes sociales.

Usos principales



En la imagen previa podemos visualizar las características generales del lenguaje en el recuadro gris.

Además vemos que describe que Lua tiene ideas similares a JS & NodeJS, con lo cual la transición de uno a otro debería ser fácil.

Algunas aplicaciones del tipo servidor pueden desarrollarse usando Luvit o Coronium LS.

Del lado izquierdo, se muestra un listado de Frameworks disponibles para el desarrollo de juegos.

Luvit

Algunas aplicaciones además de gaming es el desarrollo de servidores. Luvit es un ejemplo de estos tipos de aplicaciones.

Luvit CLI puede ser usado como una plataforma de scripting como Node Js. Puede ejecutar scripts de lua como servidores, clientes, etc.

Un ejemplo de un servidor web en Luvit que responde Hello World para todas las peticiones sería:

```lua
local http = require('http')

http.createServer(function (req, res)
  local body = "Hello world\n"
  res:setHeader("Content-Type", "text/plain")
  res:setHeader("Content-Length", #body)
  res:finish(body)
end):listen(1337, '127.0.0.1')

print('Server running at http://127.0.0.1:1337/')
```

Para probarlo hacemos:

```
> luvit server.lua
Server running at http://127.0.0.1:1337/
```

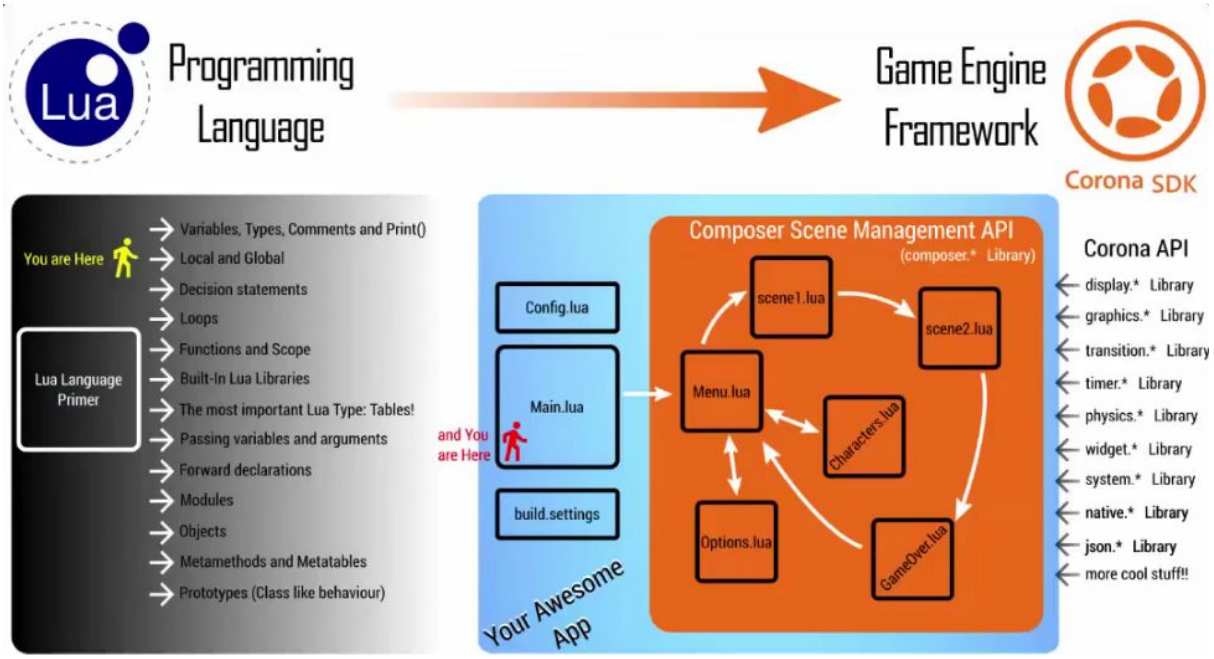
Corona

Como mencionamos previamente, el uso de Lua en gaming es lo más frecuente. Muchos juegos se desarrollan utilizando este potente framework

A continuación mostramos cómo se conectan los componentes de Lua en Corona.

Corona SDK tiene una API específica escrita en Lua que contiene varias Bibliotecas fundamentales para poder crear juegos de alta performance de manera ágil.

Una de las principales tiene que ver con el motor de física “physics” que utiliza Box2D de C++ y que por ser Lua interoperable con C pueden utilizarse juntos de una manera muy simple.



Características avanzadas

El concepto fundamental en el diseño de Lua es proporcionar meta-mecanismos para implementar algunas features en lugar de proveer estas features en el lenguaje.

Por ejemplo Lua no es un lenguaje orientado a objetos puro pero provee meta-mecanismos para implementar clases y herencia

Los meta-mecanismos mantienen el lenguaje pequeño mientras permite que la semántica sea extendida de muchas formas.

Meta tablas y meta métodos

En general las tablas en Lua tienen un set operaciones comunes. Entre ellas, se pueden agregar pares del tipo clave-valor, se puede chequear el valor asociado a una clave, se pueden recorrer todos los pares y no mucho más que eso.

Para poder utilizar toda la potencia del lenguaje tenemos las **Meta tablas**, las cuales son a su vez tablas que nos permiten agregar o cambiar comportamiento a una tabla.

Lua siempre crea las tablas sin metatablas. Para consultar la metatabla de una tabla tenemos la función **getmetatable**

```
tabla = {}
print(getmetatable(tabla))   --> nil
```

Se puede usar la función **setmetatable** para asignar o cambiar la metatabla de cualquier tabla:

```
tabla = {}
metatabla = {}
setmetatable(tabla, metatabla)
assert(getmetatable(tabla) == metatabla) → true
```

Para resumir Lua define que:

“Cualquier tabla puede ser metatabla de otra tabla.
Un grupo de tablas relacionadas puede compartir una metatabla común.

Además un tabla puede ser su propia metatabla"

Los **Meta métodos** son funciones que poseen las metatablas. Son los que en verdad proveen el comportamiento a las tablas. Existen los meta métodos como los **aritméticos**, de **relación**, y otros muy usados como **__tostring**, **__call**, **__index** y **__newindex**.

Meta métodos aritméticos

Se puede definir qué hacen los operadores aritméticos cuando se utilicen con tablas.
Por ejemplo, se puede definir que el operador ('+') represente la unión y que el operador ('*') represente la intersección.

Para cada operador aritmético hay un campo en la metatabla.
La lista completa es la siguiente:
__add (para sumar), __mul (para multiplicar), __sub (para resta), __div (para división), __unm (para negación), and __pow (para potencia).

Por ejemplo, en el siguiente programa implementamos el operador suma como la unión de dos tablas, s1 + s2 que da como resultado la tabla s3

```
Set = {}

function Set.new (tabla)
  local set = {}
  setmetatable(set, Set.meta)
  for _, v in ipairs(tabla) do set[v] = true end
  return set
end

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

Set.meta = {}
Set.meta.__add = Set.union

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}

s3 = s1 + s2
print(s3)  --> {1, 10, 20, 30, 50}
```

Estos meta métodos admiten la ejecución de tipos mixtos, donde se podría aplicar por ejemplo la suma de una tabla y un número. En estos casos se invocaría al metamétodo __add y el comportamiento sería el definido por el programador.

Supongamos que en el ejemplo anterior queremos que si se intenta sumar una tabla con algo que no sea otra tabla arroje un error:

```
function Set.union2 (a,b)
  if getmetatable(a) ~= Set.meta or
          getmetatable(b) ~= Set.meta then
          error({code=200, msg= "Intenta unir un elemento table>Set con otro tipo de elemento
(type(a)=".. type(a) .. ", type(b)=" ..type(b)..")"})
  end
  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

local status, err =  pcall(function()
s = Set.new{1,2,3}
Set.meta.__add = Set.union2
s = s + 8
end)

if not err then
  print('TODO BIEN')
else
  print('OCURRIO UN ERROR COD:' .. err.code .. ' "' .. err.msg .. '"')
end\
```

En este ejemplo además se puede ver la manera en la que se pueden manejar los errores en Lua. Para hacer el throw de un error se utiliza la función error(). Esta función puede recibir cualquier tipo de variables. En el ejemplo se muestra como se puede hacer el throw de una tabla que tiene como elementos el código del error y el mensaje.

Para hacer el catch de la excepción usamos pcall (protected call). Esta función recibe como parámetro otra función que encapsula el código a proteger y devuelve dos variables: status, que puede ser true si todo salió bien, o false en caso de error más el mensaje de error definido por el usuario, en este caso el mensaje de error es la tabla que contiene el código del error y el mensaje.

Meta métodos de relación

Se pueden utilizar otros operadores relacionales a través de metametodos:
__eq (equality), __lt (less than), and __le (less or equal).

A diferencia de los meta métodos aritméticos, los relacionales no soportan tipos mixtos. Si se trata de comparar una tabla con un número, Lua arroja un error, igualmente cuando se quiera comparar dos tablas con distintos meta métodos de orden.

Una comparación por el igual nunca arroja error, pero si dos objetos tienen distinto meta método (aunque sean iguales) retorna false sin llamar a ningún meta método.

En el siguiente ejemplo, se define el comportamiento del operador menor o igual (<=)

```
Set.meta.__le = function (a,b)
  for k in pairs(a) do
          if not b[k] then return false end
  end
  return true
end

s1 = Set.new{2, 4}
s2 = Set.new{4, 10, 2}
print(s1 <= s2)             --> true
```

Meta métodos de acceso a tablas

Lua ofrece una manera de cambiar el comportamiento de las tablas en dos casos: la consulta y la modificación de campos ausentes en una tabla

__index

Si quisiéramos acceder al valor de un campo en una tabla que no existe, el resultado sería nil.
Pero lo que Lua hace luego de verificar que no exista en la tabla es buscar por el __index meta método: si no existe, entonces retorna nil; pero si existe, es meta método es el que provee el resultado.

El uso de una tabla como el meta método __index es una forma fácil y simple para implementar herencia simple.

Obs: se podría usar una función en lugar de una tabla para herencia simple, pero una función es más costosa. De todas formas, para otros casos más complejos, probablemente usemos funciones ya que las mismas proveen mayor flexibilidad: por ejemplo permiten implementar herencia múltiple, chaching, etc.

Obs: Cuando queramos acceder a un campo de la tabla sin invocar al meta método __index se puede usar la función **rawget(t, i)**

__newindex

Antes vimos que __index sirve para acceder a los campos de una tabla.
Para actualizar los campos, se accede al meta método __newindex

Cuando se asigna un valor a un índice inexistente, el intérprete busca al __newindex: si existe, el intérprete hace la llamada a este en lugar de asignar la asignación del índice a la tabla.

Obs: hay que tener en cuenta, que como en el caso del meta método __index, si __newindex es en sí una tabla, el intérprete realiza la asignación a esa tabla en lugar de la original.

Existe una función que permite saltear al meta método: la llamada a **rawset(t, k, v)** setea el valor v en la clave k de la tabla t sin invocar ningún meta método.

La combinación de __index y __newindex permite en Lua, desde tener tablas de sólo lectura con valores default a herencia para OOP.

Para ejemplificar el uso de estos meta métodos vemos cómo se podría hacer el tracking de las consultas y modificaciones hechas a una tabla:

```
local index = {}

function track (tabla)
  local proxy = {}
  proxy[index] = tabla
  setmetatable(proxy, {
          __index = function (t,k)
          print("*ACCEDO AL ALEMENTO " .. tostring(k))
          return tabla[index][k]
          end,

          __newindex = function (t,k,v)
          print("*ACTUALIZO EL ELEMENTO " .. tostring(k) ..
```

```
                    " A " .. tostring(v))
            tabla[index][k] = v
            end
  })
  return proxy
end

t1 = {}
t1 = track(t1)

t2 = {}
t2 = track(t2)

t1[1] = 'TABLA1: ELEM UNO'
print(t1[1])

t2[1] = 'TABLA2: ELEM UNO'
print(t2[1])

t1[2] = 'TABLA1: ELEM DOS'
print(t1[2])

t2[2] = 'TABLA2: ELEM DOS'
print(t2[2])
```

Además podemos utilizar el meta método para definir tablas de sólo lectura. Donde tendríamos que bloquear el intento de actualizar cualquier valor de la misma. Por ejemplo, si tuviéramos una tabla que contenga los días de la semana:

```
function readOnly (t)
  local proxy = {}
  local mt = {
            __index = t,
            __newindex = function (t,k,v)
            error("intento de modificar una tabla de solo lectura", 2)
            end
  }
  setmetatable(proxy, mt)
  return proxy
end

dias = readOnly{"Domingo", "Lunes", "Martes", "Miercoles",
            "Jueves", "Viernes", "Sabado"}

print(dias[1])
dias[2] = "Monday" → intento de modificar una tabla de solo lectura
```

Programación orientada a objetos

Para soportar OOP Lua usa tablas.

Las tablas, al igual que los objetos, tienen estado propio y tiene identidad, además su ciclo de vida no depende de quién los crea o dónde y por último tanto objetos como tablas pueden tener su propio comportamiento.

Cuando se quiera implementar tablas como objetos se puede utilizar un parámetro extra en las funciones (métodos), ese parámetro le dice a la función sobre qué objeto debe operar. Usualmente se llama a este parámetro self o this.

Clases

Lua no posee el concepto de clases, cada objeto define su propio comportamiento. Pero esto se puede implementar siguiendo los conceptos de los lenguajes basados en prototipos (Self, NewtonScript) En estos lenguajes los objetos no tienen clases. En lugar tienen prototipos, que son objetos a los cuales el primer objeto consulta cualquier operación que no sepa ejecutar él mismo. Para esto usamos, lo que vimos antes de meta métodos, en particular __index.

Para mostrar un el uso básico definimos la clase Cuenta, que tiene métodos para consultar el saldo, depositar y extraer dinero:

```
Cuenta = {}
function Cuenta:new(o)
   o = o or {_saldo=0} -- si no se provee un objeto asigna uno con los valores default
   setmetatable(o, self)
   self.__index = self
   return o
end

function Cuenta:saldo()
   return self._saldo
end
function Cuenta:extraer(v)
   print('Extraigo '.. v)
         self._saldo = self._saldo - v
end
function Cuenta:depositar(v)
   print('Deposito '.. v)
   self._saldo = self._saldo + v
end
```

```
c2 = Cuenta:new{_saldo=100}
c2:depositar(200.00)
print('Saldo '..c2:saldo())
c2:extraer(100.00)
print('Saldo '..c2:saldo())
```

Herencia

Si ahora quisiéramos definir un tipo de cuenta especial que permitiera extraer hasta cierto límite podríamos heredar de Cuenta y re definir el método extraer para que cuando se exceda el límite arroje una excepción.

```
CuentaEspecial = Cuenta:new()

function CuentaEspecial:extraer (v)
  if v - self._saldo >= self:getLimit() then
          error ("fondos insuficientes.. El saldo negativo supera el limite de " .. tostring(self:getLimit()))
  end
  self._saldo = self._saldo - v
end

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

s = CuentaEspecial:new{limit=1000.00}
print(s)
s:depositar(100.00)
print(s)
s:extraer(500.00)
print(s)
s:extraer(700.00)
print(s)
```

Corrutinas

Resumiendo brevemente el concepto de Corrutina podemos decir que es similar a un hilo, es decir, que es una línea de ejecución con su propio stack, sus propias variables locales y su propio puntero para las instrucciones pero además comparte las variables globales y cualquier otro elemento con las demás corrutinas.

La principal diferencia que existe entre los *hilos* y las *corrutinas* es que en un programa que maneja hilos, estos corren de manera concurrente, las **corrutinas** por otro lado son colaborativas. Entonces, cuando el programa corre una corrutina sólo corre una (no puede correr dos al mismo tiempo), y la suspensión de esta sólo es lograda si se suspende de manera explícita.

Todas las funciones relacionadas con las corrutinas en **Lua** se encuentran en la tabla de corrutinas.

La función **create(f)** nos permite crearlas. Esta función recibe un argumento y es la función con el código que la corrutina correrá. El valor de retorno es del tipo **thread**, el cual representa la nueva corrutina creada.

Al ser creada, empieza en el estado **suspendido**, es decir que la corrutina no corre de manera automática cuando es creada por primera vez.

Para consultar el estado de una corrutina se ejecuta la función **status(co)** que devuelve alguno de estos valores: suspended, dead, running

Para poder a correr una corrutina (o reiniciar su ejecución) debemos usar la función **resume(co),** la cual internamente cambia el estado de la corrutina pasada por parámetro de suspendida a corriendo.

Ahora, para lograr todo este mecanismo de colaboración, la función en la cual recae gran parte de la responsabilidad es **yield().** Esta función es la que permite suspender una corrutina que se encuentra ejecutándose para poder resumir su funcionamiento luego.

Ejemplo básico de uso:

```
co = coroutine.create(function()
  for i=1,10 do
    print(“resumiendo corrutina”, i)
    courutine.yield()
  end
end)
courutine.resume(co)
courutine.resume(co)
courutine.resume(co)
courutine.resume(co)
```

Esta función corre hasta el primer yield, y sin importar que tengamos un ciclo for, la misma solo imprimirá de acuerdo a tantos resume tengamos para nuestra corrutina. La explicación básica paso a paso sería: En el flujo principal se crea la corrutina. Luego se llama a la función resume y comienza la ejecución del ciclo for. La primera instrucción dentro del for imprime el número de ciclo i y luego llama al yield, lo cual hace que se detenga y pase el control al flujo principal, que vuelve a hacer una llamada a resume, con lo cual vuelve a ejecutarse siguiente ciclo de la corrutina con un valor distinto de i. La salida de esta ejecución será:

```
resumiendo corrutina 1
resumiendo corrutina 2
resumiendo corrutina 3
resumiendo corrutina 4
```

Estadísticas

Durante estos últimos años, Lua no ha logrado subir en los rankings más importantes.

Position	Programming Language	Ratings
21	SAS	0.918%
22	R	0.911%
23	D	0.911%
24	Dart	0.601%
25	Transact-SQL	0.558%
26	Scratch	0.533%
27	Lua	0.514%
28	ABAP	0.497%
29	Fortran	0.480%
30	F#	0.460%
31	Scala	0.434%
32	Lisp	0.404%
33	COBOL	0.396%
34	Logo	0.391%
35	Ada	0.325%
36	ML	0.292%
37	PL/I	0.281%
38	Rust	0.259%
39	Prolog	0.245%
40	Kotlin	0.237%
41	TypeScript	0.230%
42	Haskell	0.218%
43	Haxe	0.216%
44	Julia	0.214%

ranking TIOBE

Si bien su uso se mantiene constante desde el año 2013 hacia el 2019, lo cual implica que el lenguaje no está desapareciendo, tampoco es que se proyecte que eso vaya a cambiar de manera positiva en el corto plazo.

Según el ranking TIOBE, Lua se encuentra en el puesto 27 de los lenguajes más utilizados del mercado. Esta posición se ubica bastante por debajo de otros lenguajes que soportan scripting como python, javascript, php, pero supera ampliamente a lenguajes no tan potentes como TypeScript que se encuentra en el puesto 40.

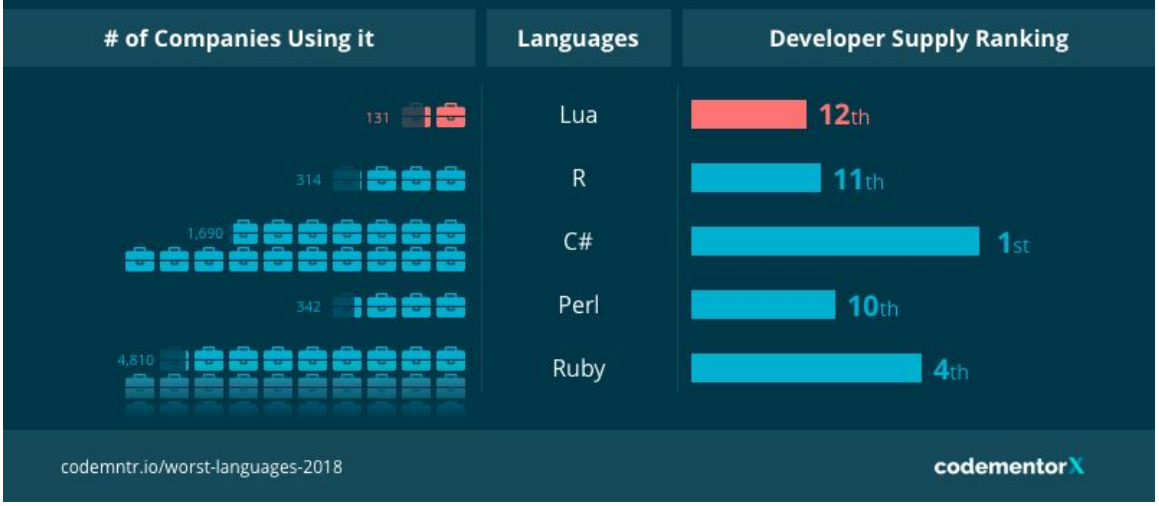
Según PYPL (PopularitY of Programming Language Index) la popularidad de Lua se mantiene constante, alrededor del 0.37% de la comunidad lo consulta en el 2019.

En otros sitios como Codementor lo catalogan como uno de los lenguajes en los cuales no se debería dedicar tiempo en aprender. Algo que tiene sentido si nos basamos en los dos rankings anteriores.

Según las estadísticas de la misma fuente para el 2018, el mercado laboral para profesionales de Lua no es muy amplio como se puede apreciar en la siguiente imagen

Lua Job Market

What Not to Learn in 2018



codemntr.io/worst-languages-2018

codementorX

Referencias

- https://www.lua.org/pil/
- https://www.tutorialspoint.com/lua/
- http://lua-users.org/wiki/
- https://coronalabs.com/
- https://box2d.org/about/
- https://luvit.io/
- https://www.tiobe.com/tiobe-index/
- http://pypl.github.io/PYPL.html
- https://www.codementor.io/blog/worst-languages-to-learn-3phycr98zk
- https://www.codementor.io/blog/worst-languages-2019-6mvbfg3w9x