# **Pregunta 01**

Escoja algún lenguaje de programación de alto nivel y de propósito general cuyo nombre empiece con la misma letra que su apellido (por ejemplo, si su apellido es “Rodríguez”, podría escoger “Ruby”, “Rust”, “R”, etc.).

(a) Dé una breve descripción del lenguaje escogido.

i. Enumere y explique las estructuras de control de flujo que ofrece.

ii. Diga en qué orden evalúan expresiones y funciones.

A. ¿Tiene evaluación normal o aplicativa? ¿Tiene evaluación perezosa?

B. La evaluación de argumentos/operandos se hace de izquierda a derecha, de

derecha a izquierda o en un orden arbitrario.

(b) Implemente los siguientes programas en el lenguaje escogido:

i. Considere la siguiente función:

Definimos la función como la cantidad de aplicaciones consecutivas de que se deben hacer sobre , hasta que el resultado sea 1.

Por ejemplo:

Por lo tanto, count(42) = 8.

Escriba un programa que, dado un entero , calcule count(n).

ii. Implemente el algoritmo Mergesort y explique los detalles de su implementación.

Nota: Explicar los detalles no implica traducir línea por línea a lenguaje natural, sino explicar el funcionamiento a grandes rasgos y las decisiones de implementación.

# **Respuesta 01**

a.i

1. **Secuenciación**

Lua ejecuta instrucciones en orden secuencial dentro de un **chunk** o un **block**. Un **chunk** es la unidad de ejecución (un archivo) y un **block** agrupa instrucciones (selecciones, repeticiones, etcétera). En Lua no existe un operador dedicado de secuenciación como **;** en C, pero podemos separar instrucciones por **;** si están en la misma línea o por saltos de línea.

a

b

1

2

Salida

print("a");print("b")

print("1")

print("2")

Lua

1. **Selección**

Lua tiene la estructura de selección **if … then … elseif … else … end.** No existe **switch-case** nativo en la sintaxis del lenguaje ni tampoco **pattern-matching**.

dos

Salida

local x = 2

if x == 1 then

  print("uno")

elseif x == 2 then

  print("dos")

else

  print("otro")

end

Lua

1. **Repetición**

Lua implementa ambas categorías de repetición: repetición determinada y repetición indeterminada, reflejadas en las estructuras **for k, v, x do … end** y **while expr do … end**, respectivamente. Además, existe la estructura **for k,v in iter do … end** que consume un iterador.

1

2

3

hola

hola

hola

hola

1

3

5

a 1

b 2

Salida

-- indeterminada

local i = 1

while i <= 3 do

  print(i); i = i + 1

end

-- repeat-until (ejecuta al menos una vez)

repeat

  print("hola")

  i = i - 1

until i == 0

-- determinada

for j = 1, 5, 2 do

  print(j)  -- 1,3,5

end

-- for generico (iterator)

for k,v in pairs({a=1,b=2}) do print(k,v) end

Lua

1. **Concurrencia**

Lua no ofrece ejecución de instrucciones simultáneas, pero sí ofrece hilos cooperativos llamados **coroutines** que son independientes al hilo principal. Estos hilos cooperativos tienen sus propias pilas y comparten variables globales, que pueden **yieldar** su ejecución y ser retomadas en un punto posterior de la ejecución.

inicio

true

despues del yield

true

Salida

co = coroutine.create(function ()

  print("inicio")

  coroutine.yield()

  print("despues del yield")

end)

print(coroutine.resume(co))

print(coroutine.resume(co))

Lua

1. **Excepciones y especulaciones**

Lua maneja errores a través de **error**, **pcall**, **xpcall** (captura de errores, llamadas en modo protegido), pero no tiene soporte transaccional incorporado, deshaciendo automáticamente instrucciones si una condición no se cumple. Cualquier rollback debe implementarse explícitamente en el código (haciendo un snapshot)

function may\_fail(x)

  if x == 0 then error("boom") end

  return 10/x

end

local ok, res = pcall(may\_fail, 0)

print(ok, res)  -- false, "boom"  (pcall atrapó el error)

Lua

false test.lua:2: boom

Salida

1. **No determinismo**

Lua no ofrece backtracking automático ni undo implícito del estado del programa en la semántica base del lenguaje. De forma similar, si queremos simular un comportamiento no determinista, se debe programar explícitamente. Es decir, debemos guardar el estado o restaurarlo cumpliendo determinada condición.

local snapshot = { x = x, t = table.unpack(t) }  -- simplificación

-- intentar una acción; si falla, restaurar desde snapshot

Lua

En este código, guardamos el estado del programa hasta el punto de que comenzamos la ejecución de una acción que podría fallar. Según la sintaxis y semántica del lenguaje, creamos un **table** que contiene el par **x:x** (es decir, almacenamos el valor de **x** en la llave **x** de la tabla) y t es un **table** que, con el método **unpack**, desincorpora todos los elementos de **t**, retornando cada uno de los elementos en el **table** como valores separados. De esta manera almacenamos el estado del programa.

1. **Asignación**

Lua no tiene transparencia referencial. Es decir, hay casos bordes y es posible cambiar el estado de las variables del programa que no pertenezcan al alcance de una subrutina. Además, Lua usa un modelo mixto: tiene un modelo de valor para unos tipos y un modelo de referencia para otros. Por ejemplo, utiliza un modelo de valor para primitivos como números, strings y booleanos, pero para objetos como **tables**, funciones y corutinas usan el modelo de referencia.

1

2

true

false

hola

adios

2

Salida

-- enteros son valores

s = 1; t = s; t = 2; print(s); print(t)

-- boolean son valores

s = true; t = s; t = false; print(s); print(t)

-- strings son valores

s = "hola"; t = s; t = "adios"; print(s); print(t)

-- tables son referenciadas

a = {1}; b = a; b[1] = 2; print(a[1])

Lua

1. **Ortogonalidad**

Un lenguaje es ortogonal si las características del lenguaje pueden ser usada en cualquier combinación y cualquier combinación tiene sentido y significado consistente. Por ejemplo, una de estas características es el uso del operador **#**, que retorna la longitud de un **table**. Sin embargo, este operador puede fallar cuando el **table** tiene “huecos” (es decir, contiene un valor nil). Por ejemplo.

2

Salida

t = {1, 2, nil}

print(#t)

Lua

La longitud de un **table** va a depender de un objeto llamado **border**. El **border** debe ser distinto de **nil** para ser considerado en el **table** y así determinar su longitud. Es decir, el operador **#** no tiene un comportamiento consistente de mostrar la longitud del **table** cuando este contiene un hueco al final.

a.ii.A. Según el manual, Lua tiene evaluación aplicativa. Es decir, las expresiones se evalúan antes de ser pasadas como argumentos a una función. Como la evaluación perezosa es un tipo de evaluación normal y Lua no tiene evaluación normal, entonces concluimos que Lua no tiene evaluación perezosa.

a.ii.B. Asimismo, tenemos que todas expresiones son evaluadas de izquierda a derecha. Para el caso de las llamadas a funciones, los argumentos son evaluados de izquierda a derecha antes de la llamada. Y, de forma similar, para los **table constructors** los campos de expresión son evaluados de izquierda a derecha. El manual no declara la existencia de operadores con orden de evaluación arbitrario.

function a() print("a"); return 1 end

function b() print("b"); return 2 end

print(a() + b()) --Ejecutándose, se evalúa primero a, luego b

Lua

Cabe destacar, adicionalmente, que Lua tiene evaluación cortocircuito de expresiones booleanas. Es decir, los operandos **and** y **or** usan evaluación corto circuito y, al momento de detectar una expresión como **false**, deja de evaluar el resto de la expresión o continúa hasta encontrar una expresión que evalúe a **true**.

function foo() print("foo evaluado"); return true end

print(false and foo())  -- foo nunca se evalúa

print(true  or  foo())  -- foo nunca se evalúa

print(true  and foo())  -- foo sí se evalúa

Lua

b.i

Debido al pequeño tamaño del código, lo adjuntamos en este documento. Sin embargo, puede verificar el repositorio de esta entrega en el siguiente link para leer el código fuente:

[**https://github.com/Arcargotte/Examen-2-Lenguajes-de-Programacion**](https://github.com/Arcargotte/Examen-2-Lenguajes-de-Programacion)

function Dist(n)

    if (n == 1) then return 0 end

    if (n % 2 == 0) then return 1 + Dist(n / 2) end

    if (n % 2 == 1) then return 1 + Dist(3 \* n + 1) end

end

print(Dist(42))

Lua

Este código fuente no es más que la definición directa de la función

Sin embargo, combina la lógica de la función **Dist** para que, por cada llamada recursiva, sume 1 al resultado final. Así, cuando llega al caso base y la función finalmente evalúa a 1, retorna 0 (pues, cuando llega a 1 no cuenta según la definición de **Dist**), y durante el retorno, suma cada una de las llamadas hechas a la función y devuelve el resultado final.

b.ii

Debido al pequeño tamaño del código, lo adjuntamos en este documento. Sin embargo, puede verificar el repositorio de esta entrega en el siguiente link para leer el código fuente:

[**https://github.com/Arcargotte/Examen-2-Lenguajes-de-Programacion**](https://github.com/Arcargotte/Examen-2-Lenguajes-de-Programacion)

function Split(t)

  local mid = math.floor(#t / 2)

  local left, right = {}, {}

  for i = 1, mid do

    left[i] = t[i]

  end

  for i = mid+1, #t do

    right[#right+1] = t[i]

  end

  return left, right

end

function MergeSort(A)

    if (#A == 1) then

        return A

    end

    local left, right = Split(A)

    return Merge(MergeSort(left), MergeSort(right))

end

function Merge(A, B)

    local merged = {}

    local i,j = 1, 1

    for k = 1, #A + #B do

        if j > #B or (i <= #A and A[i] <= B[j]) then

            table.insert(merged, A[i])

            i = i + 1

        else

            table.insert(merged, B[j])

            j = j + 1

        end

    end

    return merged

end

function PrintList(A)

    for i = 1, #A do

        io.write(A[i])

        io.write(" ")

    end

    io.write("\n")

end

PrintList(MergeSort({3,6,8,1,7}))

Lua

Este código fuente es una implementación del algoritmo **MergeSort** hecho en Lua. Se trata, pues, de una implementación *línea a línea* que sigue la definición del algoritmo sin nada destacable a añadir. **Merge** es una subrutina de la rutina recursiva **MergeSort**, que dado un arreglo **A** de longitud mayor a **1**, la divide en dos mitades **left** y **right**, considerando el medio como , donde es la longitud del arreglo A. Esto lo hace gracias a la rutina **Split**, que se encarga de tomar un arreglo **t** y dividirlo en arreglos **left** y **right**. Dados estos dos arreglos, que se suponen ya ordenados de menor a mayor, la subrutina **Merge** se encarga de combinar ambos arreglos, asegurándose que el resultado final sea el arreglo **A** ordenado de menor a mayor. Para hacer esto, crea un arreglo cuya magnitud es la suma de las longitudes de **left** y **right** y, elemento a elemento de **left** y **right**, los va comparando para determinar el menor entre un par de elementos e insertarlo en el arreglo, que estará ordenado para el final de la rutina.

Lo más destacable de esta implementación de **MergeSort** es la experiencia del programador en Lua: debido a la naturaleza simple del lenguaje, no existen procedimientos para imprimir la lista en la salida estándar ni tampoco una manera sencilla de dividir un arreglo en **slices** en función de subíndices (o sea, una manera similar en lenguajes como Python, donde podemos conseguir la mitad de un arreglo **A** como **A[: len(A) // 2])**. El programador debe hacer sus propias funciones para imitar estos comportamientos que proveen otros lenguajes de base. Para ilustrar este último punto, basta con ver la implementación de **Split** y **PrintList**, que se encarga de obtener dos **slices** de **A**, que serán usados como los arreglos **left** y **right** en **MergeSort**, y de imprimir el arreglo **A** ya ordenado para verificar resultados.

# **Referencias bibliográficas**

[1] I Pou, R. (2016). *Lua 5.4 Reference Manual*. Lua.org. Sección 3.4.7: “The Length Operator”. Disponible en <https://www.lua.org/manual/5.4/manual.html#3.4.7>