Introducción a Lua

Tiempo estimado: 50min

Redis permite la realización de consultas complejas, en el propio servidor, mediante Lua. Ésta es la primera lección, de varias dedicadas a este lenguaje. Es muy importante conocerlo para poder sacar todo el jugo a Redis.

Ésta es sin duda la lección más larga del curso. Comienza introduciendo el lenguaje y su intérprete oficial. A continuación, se describe los elementos del lenguaje como, por ejemplo, los comentarios, las variables, los tipos de datos, las sentencias de control de flujo, los bloques y las funciones.

Al finalizar la lección, el estudiante sabrá:

- Cómo instalar el intérprete para aprender Lua.
- Cómo definir expresiones Lua.
- Cómo definir sentencias de control de flujo.
- Cómo definir funciones.

Introducción

Al igual que otros sistemas de gestión de bases de datos, Redis proporciona un lenguaje con el que escribir *scripts* de servidor. Así por ejemplo, Cassandra utiliza CQL; SQL Server, T-SQL; PostgreSQL principalmente SQL y PL/pgSQL; y CouchDB y MongoDB, JavaScript. Lua es el lenguaje de programación elegido por Redis para escribir *scripts* nativos. Incorpora un intérprete de este lenguaje, con el que poder realizar consultas más complejas dentro del propio servidor de bases de datos.

Fue creado en 1993 por Roberto Ierusalimschy, Luis Henriquez de Figueiredo y Waldemar Celes de la Pontifícia Universidade Católica do Rio de Janeiro. Su sitio web oficial es lua.org.

Se diseñó para facilitar la extensión de programas de software. Pudiéndose integrar fácilmente con ellos. Se utiliza principalmente en el desarrollo de juegos y como lenguaje de *scripting* nativo en productos como Adobe Photoshop Lightroom, Aerospike, Angry Birds, Apache HTTP Server, HAProxy, Nginx, Redis, SimCity, VLC y World of Warcraft.

Sus principales características son:

- Es un lenguaje de propósito general (*general-purpose language*). Esto significa que con él se puede escribir programas o *scripts* en un abanico de dominios muy grande.
- Es un lenguaje tipado dinámicamente (*dynamically typed language*), esto quiere decir que cuando se declara una variable, al igual que en JavaScript o Python, no hay que indicar el tipo de valores que puede almacenar. Una variable puede contener valores de cualquier tipo de datos.
- Es un lenguaje interpretado (*interpeted language*), lo que implica que es analizado y compilado en tiempo de ejecución.
- Tiene gestión automática de memoria. Facilitando la programación, al no tener que preocuparnos por liberar la memoria ocupada por objetos que ya no se estén usando.
- Es un lenguaje multiplataforma (*cross-platform language*), pudiéndose utilizar en Android, Linux, (como, por ejemplo, Debian, Ubuntu o Raspbian), Mac OS o Windows, así como en dispositivos móviles, tabletas, ordenadores personales, servidores, Raspberry Pis, etc.
- Es un lenguaje extensible (*extensible language*). Se puede desarrollar módulos y paquetes de Lua para su reutilización en otros componentes, programas o *scripts*.
- Es simple y pequeño, facilitando así su aprendizaje.

Intérprete de Lua

Aunque lo hemos indicado al comienzo de la lección, no está de más hacer hincapié de nuevo. Lua es un lenguaje interpretado (*interpeted language*), lo que implica que es analizado y compilado en tiempo de ejecución.

El programa que analiza, compila y ejecuta un programa de Lua se conoce formalmente como motor de Lua (*Lua engine*). Se encuentra implementado en C, en forma de biblioteca para su fácil integración en aplicaciones, principalmente, en aquellas escritas en C/C++. También se dispone de un intérprete de línea de comandos con el que poder ejecutar *scripts* de Lua sin necesidad de tener que utilizar una aplicación que lo lleve integrado.

Generalmente, no es muy recomendable aprender Lua con el intérprete incorporado a una aplicación. Por ejemplo, Redis lleva incorporada la librería de Lua, permitiendo así la ejecución de *scripts* con este lenguaje en el servidor de bases de datos. Pero aprender Lua con este intérprete es tedioso, muy tedioso. Por esta razón, recomendamos utilizar lua, el intérprete de línea de comandos oficial, el cual, como no podía ser de otra manera, lleva incorporada la librería de Lua.

Instalación del intérprete lua

Puede descargar el intérprete de Lua de su sitio web, lua.org.

En sistemas Debian como Ubuntu, puede instalar el paquete lua5.1 con apt:

```
$ sudo apt install -y lua5.1
```

En el momento de escribir estas líneas, abril de 2017, Redis utiliza la versión 5.1 del lenguaje. Estando disponible ya la versión 5.3. Si le gusta Lua y desea jugar con este lenguaje, le recomendamos instale la versión más reciente.

Una vez instalado, puede acceder al intérprete interactivo mediante el comando lua:

```
$ lua
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
>
```

Comentarios

Un comentario (*comment*) es un fragmento de texto que no tiene ningún significado especial para el lenguaje ni el programa. En Lua, los comentarios son hasta fin de línea o multilínea. Los comentarios hasta fin de línea comienzan con dos guiones (--) y acaban al finalizar la línea. En cambio, un comentario multilínea es aquel que puede expandirse, si se desea, en múltiples líneas. Este tipo de comentarios se puede expresar de varias formas, siendo la delimitada por --[[y]] la más usada.

He aquí unos ejemplos:

```
1 + 2 + 3 --comentario hasta fin de línea
--[[ Éste es un comentario
que se expande en múltiples líneas. ]]
```

Variables

Un contenedor de datos (*data container*) es un recipiente o lugar para almacenar un determinado valor. Atendiendo a si podemos cambiar el valor almacenado a lo largo del programa, se distingue entre variable y constante. Una variable es un contenedor al que podemos cambiar su valor a lo largo del programa. En cambio, una constante (*constant*), una vez asignado el valor, no podemos cambiarlo. En Lua, sólo existe el concepto de variable.

Toda variable debe tener, por un lado, un nombre (*name*) para identificarla del resto y, por otro lado, el dato almacenado, conocido formalmente como valor (*value*). Cuando deseemos acceder a su valor, bastará con indicar el nombre que le hemos asignado a la variable. Al igual que otros lenguaje como C/C++ y JavaScript, Lua es sensible a mayúsculas y minúsculas.

Los nombres o identificadores se expresan como una secuencia de uno o más caracteres. Pueden tener letras, dígitos y el signo de subrayado (_). Pero nunca pueden comenzar por un dígito. He aquí unos ejemplos ilustrativos: edad, telCasa, tel_trabajo, x123, _longitud, etc.

Tipos de datos

Los lenguajes de programación permiten trabajar con valores de distintos tipos como, por ejemplo, números, cadenas de texto, booleanos, etc. Definimos, pues, un tipo de datos (*data type*) como un posible conjunto de valores que podemos utilizar en el programa. Lua proporciona varios tipos de datos. Vamos a presentarlos.

Tipo de datos string

El tipo string representa un texto en forma de secuencia de caracteres. El conjunto de textos que se puede crear es infinito y está formado por todas las posibles combinaciones de caracteres de cualquier tamaño.

Literales de texto

Un valor literal (*literal value*) o simplemente literal es una unidad léxica que denota algo tal cual como, por ejemplo, un número o una cadena de texto. Los literales de los tipos de datos de Lua son muy parecidos a los de C/C++, Java, JavaScript o Python.

Podemos expresar los valores literales de tipo cadena de dos maneras. Por un lado, mediante una secuencia de cero, uno o más caracteres delimitados por comillas simples (') o dobles ("). Otra posibilidad es delimitar el literal entre dos pares de corchetes ([[y]]), permitiendo expandir la cadena literal en varias líneas. Cuando se usa esta segunda opción, si tras el inicio del literal aparece un salto de línea, el intérprete lo ignorará.

Veamos unos ejemplos ilustrativos:

```
txt = "Esto es una cadena literal"
txt = [[
  Esto es una cadena literal
  que se expande a lo largo de varias
  líneas.]]
```

Secuencias de escape

Una secuencia de escape (*escape sequence*) representa un carácter especial como, por ejemplo, el tabulador o el salto de línea. Algo que es difícil de escribir o que tiene un significado especial. La siguiente tabla presenta las secuencias de escape de Lua:

Secuencia de escape	Carácter representado
\\	Carácter \
\"	Carácter de comillas dobles (")
\'	Carácter de comillas simples (')
\0	Carácter nulo
\a	Carácter de emisión de sonido
\b	Carácter de borrado
\f	Carácter de salto de página
\n	Carácter de nueva línea
\r	Carácter de retorno de carro
\t	Carácter de tabulado horizontal
\v	Carácter de tabulado vertical
\xHH	Carácter indicado por el valor hexadecimal *HH*
\uHHHH	Carácter Unicode indicado por el valor hexadecimal *HHHH*

Es importante destacar que las cadenas literales delimitadas por corchetes ([[y]]) no interpretan las secuencias de escape. Los caracteres de escape serán interpretados como tales, no como el carácter al que representan.

Conversiones

La conversión (*casting*) es la acción mediante la cual convertimos un valor de un tipo a otro. En el caso del tipo string, se puede utilizar la función tostring():

```
function tostring(value) : string
```

Parámetro	Tipo de datos	Descripción
value	Cualquiera	Valor a convertir.

La siguiente tabla muestra cómo se convierte los datos de un tipo al tipo cadena:

Tipo de datos	Valor devuelto
nil	El texto *nil*.
number	El número en texto.
string	El propio texto.
table	El texto *table* seguido de dos puntos y su posición en memoria.
boolean	El texto true o false, según el valor booleano a convertir.
function	El texto *function* seguido de dos puntos y su posición en memoria.

Tipo de datos number

El tipo number representa un número entero o real.

Literales numéricos

Los valores enteros literales se indican mediante su secuencia de dígitos, uno detrás de otro, al igual que nos enseñaron en la escuela como, por ejemplo, 1234, 56789, -101, etc.

Los reales se especifican separando la parte decimal de la entera por un punto. Ejemplos: 12.34, -101.5, etc.

Conversiones

Para convertir un valor a un tipo number, se puede utilizar la función tonumber():

```
function tonumber(value) : number
```

Parámetro	Tipo de datos	Descripción
value	cualquiera	Valor a convertir.

La siguiente tabla muestra cómo se convierten los datos de un tipo al tipo number:

Tipo de datos	Valor devuelto
nil	El valor nil.
boolean	El valor nil.
string	El número expresado por la cadena o nil si no es un número.
table	El valor nil.
function	El valor nil.

Tipo de datos boolean

El tipo boolean representa un valor lógico verdadero o falso, estando su posible conjunto de valores formado por true o false.

Literales booleanos

Los valores literales del tipo boolean son las palabras reservadas true (verdadero) y false (falso).

Tipo de datos nil

El tipo nil almacena un único valor, en este caso, el valor nulo. Este valor se utiliza para indicar que una variable contiene el valor nulo, al igual que el null de C/C++, C#, Java y JavaScript o el None de Python.

Literal nulo

Para indicar el valor literal del tipo nil se utiliza la palabra reservada nil.

Tipo de datos table

Mediante el tipo de datos table se puede almacenar datos compuestos. Es multiforme; puede representar una lista de valores o bien un *array* asociativo. Los *arrays* asociativos son como los homónimos de Python o los objetos de JavaScript.

Literales de tabla

Los valores literales del tipo table se delimitan entre llaves ({ y }). Básicamente, tenemos dos formas:

```
--tabla vacía
{}

--lista o array
{elemento, elemento, elemento...}

--array asociativo u objeto
{clave = valor, clave = valor, clave = valor...}
```

Recuerde que con el tipo table se puede representar tanto una lista como un *array* asociativo.

Cuando deseamos representar una lista, accedida mediante índices, cada uno de los elementos se separa el uno del otro por una coma, siendo su posición en el literal su índice: 1 para el primer elemento, 2 para el segundo y así sucesivamente.

En cambio, cuando deseamos expresar un *array* asociativo u objeto, cada elemento está formado por una clave y un valor. La clave se puede indicar de dos formas distintas:

```
clave
["clave"]
```

He aquí unos ejemplos ilustrativos:

```
{nombre = "Leonard Cohen", edad = 82}
{["nombre"] = "Leonard Cohen", ["edad"] = 82}
```

Acceso a valores de tabla

Para acceder a un elemento concreto de una tabla, se utiliza el operador punto (.) o el de indexación ([]) como sigue:

```
--lista o array
tabla[índice]

--array asociativo u objeto
tabla.campo
tabla["campo"]
```

A continuación, se muestra cómo acceder a un elemento de una lista y a un campo de un *array* asociativo:

```
tags[2]
usuario.nombre
usuario["nombre"]
```

Realmente, cuando se indica el índice o el campo entre corchetes, se puede pasar una expresión, cuya evaluación indicará el índice o el campo en cuestión.

Función type()

La función type() se puede utilizar para obtener una cadena de texto que indica el tipo de un determinado valor:


```
function type(value) : string
```

Parámetro	Tipo de datos	Descripción
value	Cualquiera	Tipo de datos.

La siguiente tabla presenta los valores devueltos por la función para cada uno de los tipos de Lua:

Tipo de datos	Valor devuelto por type()
boolean	boolean
function	function
nil	nil
number	number
string	string
table	number

Expresiones

En este punto, ya sabemos cómo definir variables y cómo representar valores literales. Ahora, vamos a presentar cómo combinarlo todo para formar expresiones que pueda reconocer e interpretar el motor de Lua.

Recordemos que una proposición (*sentence*) es una unidad lingüística que enuncia algo mediante una secuencia de términos. Las proposiciones se clasifican en expresiones y sentencias, donde una expresión (*expression*) es una proposición formada por una combinación de términos que, tras ser evaluada, devuelve un valor. Principalmente, las expresiones están formadas por identificadores, operadores, valores literales y/o subexpresiones como, por ejemplo, `1 - 2 + 3` o `total = importe * (1 + impuesto)`.

Operadores

Un operador (*operator*) es un símbolo o palabra reservada que realiza una determinada operación como, por ejemplo, la suma, la multiplicación, la concatenación o la asignación. Los parámetros de los operadores se conocen formalmente como operandos (*operands*), los cuales pueden ser valores literales, variables o subexpresiones.

Todo operador tiene una aridad, una precedencia y una asociatividad.

La aridad (*arity*) indica el número de operandos que espera el operador. Teniendo en cuenta este número, se pueden clasificar en: operadores unarios (*unary operators*), un único operando; operadores binarios (*binary operators*), dos operandos; operadores ternarios (*ternary operators*), tres operandos; y operadores n-arios (*n-ary operators*), *n* operandos. Las sintaxis de los operadores unarios y binarios, respectivamente, son como sigue:

```
operador operando
operando operador operando
```

Como una expresión puede contener varios operadores, es necesario que el motor de Lua pueda determinar, sin temor a equivocarse, el orden en que debe ejecutarlos. No puede hacerlo aleatoriamente, porque el orden de los factores puede alterar el producto. Por esta razón, los operadores presentan una precedencia (*precedence*), esto es, una prioridad del operador con respecto a los demás dentro de la misma expresión. Así, los operadores con mayor precedencia se ejecutan antes que los de menor precedencia.

La siguiente tabla indica la precedencia de los operadores en Lua, de mayor a menor precedencia, junto a su aridad y asociatividad:

Operador	Aridad	Asociatividad
^	Binario	Por la derecha
not # -	Unario	Por la izquierda
* / %	Binario	Por la izquierda
+ -	Binario	Por la izquierda

..	Binario	Por la derecha
< > <= >= ~= ==	Binario	Por la izquierda
and	Binario	Por la izquierda
or	Binario	Por la izquierda

Según la tabla anterior, el operador de multiplicación (*) tiene mayor prioridad que el de suma (+). Teniendo esto en cuenta, observe los resultados de las siguientes expresiones:

```
> 4 * 5 + 6
26
> (4 * 5) + 6
26
> 4 * (5 + 6)
44
> 4 + 5 * 6
34
> 4 + (5 * 6)
34
> (4 + 5) * 6
54
>
```

Al igual que en otros lenguajes, se puede usar los paréntesis para delimitar una subexpresión, cuyo resultado será el operando del operador correspondiente.

Además de aridad y precedencia, los operadores presentan asociatividad (*associativity*), que utiliza el motor para decidir, en caso de conflicto con operadores de igual precedencia, cuál de ellos debe ejecutar primero. Una asociatividad por la derecha (*right-to-left associativity*), indica que tiene prioridad el de la derecha; en caso de asociatividad por la izquierda (*left-to-right associativity*), el izquierdo. Según la tabla de precedencia anterior, los operadores de multiplicación (*), división (/) y módulo (%) tienen igual precedencia. Además, tienen asociatividad por la izquierda, indicándose así que, en caso de aparecer juntos, el motor siempre ejecutará primero el izquierdo. Esto queda más claro mediante unos ejemplos:

```
> 4 * 5 % 6
2
> (4 * 5) % 6
2
> 4 * (5 % 6)
20
> 4 % 5 * 6
24
> (4 % 5) * 6
24
> 4 % (5 * 6)
4
>
```

Como acabamos de indicar, los dos operadores tienen igual prioridad, por lo que el motor de Lua utiliza la asociatividad para determinar qué operador evaluar primero. Como tienen asociatividad por la izquierda, entonces primero evaluará el operador de la izquierda con los dos operandos que le rodean y, a continuación, el segundo usando como su operando izquierdo el resultado del operador anterior. Así pues, 4 * 5 % 6 es lo mismo que (4 * 5) % 6 y 4 % 5 * 6 es igual que (4 % 5) * 6.

Cuando deseamos dejar claro cuál es el orden, se suele utilizar los paréntesis para delimitar subexpresiones que marcan que su contenido debe ejecutarse primero.

Operador de asignación

El operador de asignación (*assignment operator*) es aquel que asigna o fija el valor actual a una variable. Se representa mediante el símbolo igual (=). He aquí un ejemplo ilustrativo:

```
x = 1
tags[2] = "indie"
band.tags = {"indie", "alternativo"}
```

En Lua, las expresiones de asignación no hace falta finalizarlas en punto y coma (;), aunque se puede hacer. Así pues, las dos expresiones siguientes son equivalentes:


```
x = 1 + 2 * 3
x = 1 + 2 * 3;
```

Asignación múltiple

La asignación múltiple (*multiple asignment*) es aquella que permite asignar valores a varias variables mediante una única expresión de asignación. Lo que se hace es indicar como operandos izquierdos, separados por comas, los nombres de las variables y a la derecha del operador igual los valores. El primer valor se asignará a la primera variable, el segundo a la segunda y así sucesivamente. Veamos un ejemplo:

```
a, b, c = 1, 2, 3
```

No es necesario que el número de variables sea igual al de valores. Cuando esto sucede:

- Si el número de variables es menor que el de valores, todo aquel valor que no tenga variable asociada, se perderá.
- Si el número de variables es mayor que el de valores, aquella que no tenga valor asociado explícito se le asignará el valor nil.

Mediante la asignación múltiple, es muy sencillo implementar el intercambio de valores de dos variables:

```
a, b = b, a
```

Operadores aritméticos

Los operadores aritméticos (*arithmetic operators*) son aquellos que realizan operaciones aritméticas con números como, por ejemplo, la suma, la resta o la multiplicación. Según el operador, puede ser unario o binario. La siguiente tabla presenta los operadores aritméticos de Lua:

Operador	Aridad	Descripción	Ejemplo
-	Unario	Negación numérica.	-x
+	Binario	Suma	x + y
-	Binario	Resta	x - y
*	Binario	Multiplicación	x * y
/	Binario	División	x / y
%	Binario	Módulo, resto de la división	x % y

Operador de concatenación

El operador de concatenación (*concatenation operator*) es aquel que devuelve una cadena con el resultado de añadir una al final de otra. Se representa mediante el operador dos puntos (..). Cuidado los programadores de JavaScript, ¡no se usa +! Cuando un operando no es de tipo cadena, el intérprete lo convertirá a texto previamente a la concatenación.

He aquí algunos ejemplos ilustrativos:

```
> 123 .. 456
123456
> "buon" .. "giorno"
buongiorno
>
```

Operadores lógicos

Un operador lógico (*logical operator*) devuelve un resultado booleano para las operaciones *AND*, *OR* y *NOT* lógicas. Son muy utilizados en las expresiones de comparación de las sentencias if y while.

Operador	Aridad	Descripción	Ejemplo
and	Binario	AND lógico	x and y
or	Binario	OR lógico	x or y
not	Unario	NOT lógico	not x

Los operadores lógicos de Lua son similares a los de JavaScript. Presentan dos características muy importantes. Por un lado, son en corto circuito (*short-circuit*), esto significa que, si a partir del operando de la izquierda se puede deducir su resultado, no evalúa el segundo. Por otra parte, los operadores and y or no devuelven un valor booleano, sino uno de los dos operandos, atendiendo a su representación booleana. En Lua, se considera los valores false y nil como falsos; cualquier otro como verdadero, incluido el valor cero y la cadena vacía. Aquí de nuevo, mucho cuidado los programadores de JavaScript.

Así pues, tenemos lo siguiente:

Operador	**Descripción**
izquierdo and derecho	Devuelve el operando izquierdo si su representación booleana es false; en otro caso, el derecho.
izquierdo or derecho	Devuelve el operando izquierdo si su representación booleana es true; en otro caso, el derecho.
not operando	Convierte el operando a su representación booleana y, a continuación, la niega.

Analicemos más detenidamente los operadores. Primero, veamos el operador not, el de negación. Lo que hace es, dado un operando, obtener su representación booleana, esto es, true o false, y a continuación negarla:

```
> not true
false
> not false
true
> not nil
true
> not 123
false
>
```

El objetivo de la negación es obtener el *otro* valor booleano: si es true, false; y si es false, true. Recordemos que en Lua, todo valor *no* booleano es true a excepción de nil que se considera como false.

Ahora, veamos los operadores lógicos and y or. Estos operadores lo que hacen es devolver un operando u otro atendiendo a la representación booleana del primero. Así, en el caso del operador and, el operador devolverá el primer operando, sin evaluar el segundo, si su representación booleana es false; en otro caso, devolverá el segundo:

```
> true and true
true
> true and false
false
> false and true
false
> false and false
false
> 123 and 456
456
> nil and 456
nil
>
```

En cambio, el operador or hace lo contrario: si la representación booleana del operando izquierdo es true, lo devolverá, sin evaluar el segundo; en otro caso, devolverá el derecho:

```
> true or true
true
> true or false
true
> false or true
true
> false or false
false
> 123 or 456
123
```



```
> nil or 456
456
>
```

Los operadores and y or son similares a sus homólogos && y || de JavaScript.

Operadores de comparación

Un operador de comparación (*comparison operator*) es aquel que compara dos operandos entre sí para comprobar si son iguales o distintos, devolviendo un valor booleano como resultado. La siguiente tabla enumera los operadores de comparación de Lua:

Operador	Aridad	Descripción	Ejemplo
==	Binario	Igualdad	x == y
~=	Binario	Desigualdad	x ~= y
<	Binario	Menor que	x < y
<=	Binario	Menor o igual que	x <= y
>	Binario	Mayor que	x > y
>=	Binario	Mayor o igual que	x >= y

Los operadores de comparación de igualdad y desigualdad son estrictos. Comprueban si los operandos son del mismo tipo y, además, son el mismo valor. Esto se entiende mejor mediante unos sencillos ejemplos ilustrativos:

```
> "2" == "2"
true
> "2" == 2
false
>
```

Operador de longitud

El operador # se utiliza para obtener el número de caracteres de una cadena o el número de elementos de una tabla:

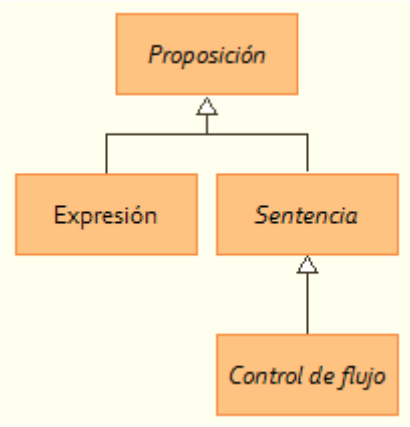
```
#Expresión
```

Ejemplo:

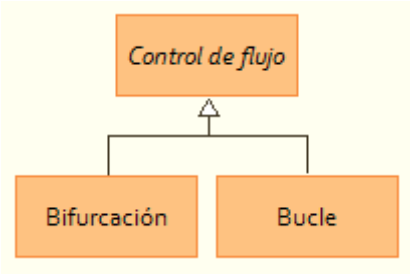
```
> #"Esto es una cadena"
18
> #{"uno", "dos", "tres"}
3
>
```

Sentencias de control de flujo

Una sentencia (*statement*) es un tipo especial de proposición mediante la cual podemos hacer determinadas cosas como, por ejemplo, definir estructuras de control o variables locales. Todos los lenguajes de programación proporcionan un conjunto de sentencias mediante las cuales poder ejecutar determinadas proposiciones del programa ante determinadas situaciones. Se conocen formalmente como control de flujo (*flow control*) o estructuras de control (*control structures*), porque permiten controlar el flujo natural de ejecución del programa.



Atendiendo a la naturaleza del control de flujo, las sentencias se pueden clasificar en bifurcaciones o bucles.



Una bifurcación (*fork*) es aquella que ejecuta un bloque de código u otro atendiendo a una determinada condición. Mientras que un bucle (*loop*) ejecuta cero, una o más veces un determinado bloque de código mientras se cumpla una determinada condición o situación.

Bifurcaciones

Una bifurcación (*fork*) es una sentencia condicional que ejecuta un bloque de código u otro atendiendo a una determinada condición. En Lua, disponemos de un único tipo de bifurcación, la sentencia if, muy parecida a la homónima de muchos otros lenguajes de programación.

Sentencia if

Esta estructura de control ejecuta un bloque de código u otro atendiendo a unas condiciones de selección. Su sintaxis es la siguiente:

```
if Expresión then
  bloque
elseif Expresión then
  bloque
else
  bloque
end
```

La sentencia comienza con una cláusula if a la que puede seguir cero, una o más cláusulas elseif y, finalmente, cero o una cláusula else. La sentencia se lee más o menos como sigue: *si* se cumple la condición, *entonces* haz lo siguiente; *si no, si* se cumple esta otra condición, *entonces* haz esto otro; *en cualquier otro caso*, ejecuta esto.

A modo de ejemplo, se muestra cómo escribir por la terminal un mensaje indicando si se trata de un número par o impar:

```
if x % 2 == 0 then
  print("Número par.")
else
  print("Número impar.")
end
```

Las sentencias de control que evalúan expresiones condicionales, como por ejemplo if, while y repeat, consideran los valores false y nil como falsos; cualquier otro, como verdadero. Los programadores de JavaScript y otros lenguajes no deben olvidar que, en Lua, la cadena vacía y el valor cero se procesan

como ciertos, no como falsos. Es muy importante recordarlo.

Bucles

Un bucle (*loop*) es una sentencia que ejecuta el mismo bloque de código mientras se cumple una determinada condición o bien se sale explícitamente mediante un return, un break, una excepción o la finalización del programa. Lua proporciona los bucles while, repeat y for.

Sentencia while

La sentencia while itera su bloque de código adjunto mientras se cumple una condición asociada. Su sintaxis es:

```
while Expresión do
  bloque
end
```

A continuación, se muestra un ejemplo en el que se enumeran los números pares entre uno dado y 30, mediante un bucle while:

```
local i = x;

while i <= 30 do
  if i % 2 == 0 then
    print(i)
  end

  i = i + 1
end
```

Sentencia repeat

La sentencia repeat es similar al bucle while, exceptuando que la condición de iteración, en vez de comprobarse al inicio, se evalúa al final de cada iteración, lo que implica que como mínimo se ejecutará una vez. Su sintaxis es como sigue:

```
repeat
  bloque
until Expresión
```

Sentencia for

La sentencia for es similar al bucle while, pero presenta secciones especiales que no presenta el segundo. Tiene varias sintaxis:

```
--for numérico
for variable = Expresión, Expresión do
  bloque
end

for variable = Expresión, Expresión, Expresión do
  bloque
end

--for genérico
for clave, valor in Expresión do
  bloque
end
```

El for numérico itera entre los números de un rango especificado. El valor inicial se indica como primera expresión, el final como segunda y, si lo deseamos, el valor de incremento se puede indicar en el tercero. Por ejemplo, para iterar por los números pares entre 0 y 100, podríamos hacer lo siguiente:

```
for x = 0, 100, 2 do
  print(x)
end
```

La variable de iteración, la que contiene el valor de la iteración actual, es local al bucle, eso significa que sólo puede accederse en su bloque o cuerpo.

Por su parte, el for genérico itera entre varios elementos como, por ejemplo, los de una tabla. Ejemplo:

```
for ix, val in ipairs({0, 2, 4, 6, 8, 10}) do
```



```
  print(val)
end
```

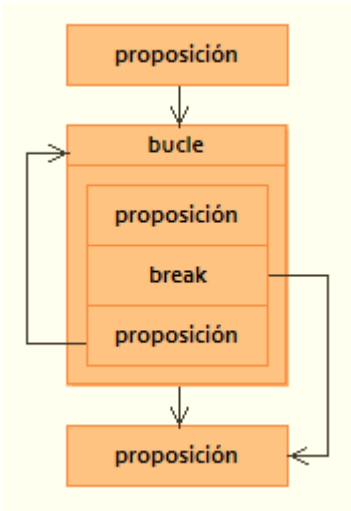
En los bucles for genéricos, es muy común utilizar las siguientes funciones para recorrer tablas:

- pairs(tab) para recorrer una tabla que representa un *array* asociativo.
- ipairs(tab) para recorrer una tabla que representa una lista.

Estas funciones reciben como argumento una tabla y devuelven un iterador, utilizado por el bucle for, para recorrer sus elementos. En cada iteración, el iterador devuelve dos elementos: el índice o nombre de campo, según corresponda; y el valor del elemento o campo.

Sentencia break

La sentencia break se utiliza para salir excepcionalmente de un bucle (while, repeat o for). Cuando el motor de Lua se encuentra con esta instrucción, abandona la ejecución del bucle y sigue con la proposición que sigue al bucle.



Su sintaxis es:

```
break
```

Ejemplo:

```
local i = x

while true do
  if i > 30 then break end
  if i % 2 == 0 then print(i) end
  i = i + 1
end
```

Bloques

Un bloque (*block*) es una secuencia de cero, una o más proposiciones de igual o distinto tipo. A diferencia de otros lenguajes como C/C++, Java o JavaScript, donde los bloques se delimitan por llaves ({ y }), en Lua, son intrínsecos al lenguaje, se crean automáticamente dentro de determinadas sentencias. Por ejemplo, en una sentencia if tenemos un bloque para la cláusula if, otro para cada elseif y otro para else.

Variables locales de bloque

En Lua, se distingue entre variables globales y locales. Una variable global (*global variable*) es aquella que se puede acceder desde cualquier parte del programa. Mientras que una variable local (*local variable*), sólo dentro del bloque en el que se define.

Toda variable *no* declarada como local es global. Así de simple. Para definir una variable como local, hay que utilizar la sentencia local:

```
local variable
local variable, variable, variable...
```



```
local variable = Expresión
local variable = Expresión, variable = Expresión, variable = Expresión
```
Cuando no se declara el valor inicial de una variable local, se inicializa a nil.

Funciones

Una función (*function*) es un objeto que representa una secuencia de proposiciones que, cuando se ejecuta, realiza una determinada operación, actividad o tarea. Es el componente principal de la programación en Lua. El uso de funciones permite dividir un programa en unidades más pequeñas, con tareas claras y bien definidas que pueden reutilizarse en cualquier momento, tanto en el mismo programa como en otros.

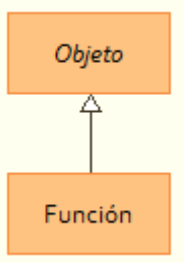
Al descomponer la programación en unidades más pequeñas y manejables, o sea, al crear funciones, se consigue código más claro y fácil de mantener. Además de evitar repetir la misma secuencia de proposiciones una y otra vez, cada vez que se desea realizar la tarea específica. Por otra parte, es más fácil realizar pruebas de unidad sobre funciones y depurarlas, que hacerlas sobre todo el código del programa en su conjunto.

Componentes de una función

En Lua, una función se divide básicamente en dos partes: la signatura y el cuerpo. Toda función tiene una signatura (*signature*) que indica su nombre y su lista de parámetros. Otros lenguajes como, por ejemplo, C/C++, C#, Java, PL/pgSQL o TypeScript, indican el tipo de valor que devolverá la función, pero Lua no. Mientras que la secuencia o bloque de proposiciones, que forman la función y realizan la tarea asignada, se conoce formalmente como cuerpo (*body*). Toda función tiene un cuerpo, aunque no tenga asociada ninguna proposición.

Objeto function

En Lua, las funciones son objetos, al igual que en JavaScript. Toda función es una instancia del tipo de datos function. Recordemos que Lua es un lenguaje basado en objetos, donde todo es un objeto. Las funciones son un tipo especial de objeto.



Como se trata de objetos, las funciones se pueden asignar a variables. Para conocer si una variable contiene o referencia a un objeto función, se puede utilizar la función type(), la cual devuelve el texto function:

```
> type(suma)
function
>
```

Definición de funciones

Una definición de función (*function definition*) es una sentencia que declara un objeto función. Se puede definir funciones mediante el operador function o la sentencia function. Veamos cada forma detenidamente.

Definición mediante la sentencia function

La sentencia function define y crea una nueva función, esto es, un nuevo objeto función. Su sintaxis es como sigue:

```
--función global
function nombre([listaParámetros])
  cuerpo
```



```
end

--función local
local function nombe([listaParámetros])
  cuerpo
end
```

Las funciones globales son accesibles desde cualquier punto del programa. Mientras que las locales sólo dentro del bloque en el que se definen.

Veamos un ejemplo ilustrativo. La siguiente función realiza la suma de dos valores, los cuales se pasan como argumentos en el momento de la llamada a la función:

```
function suma(x, y)
  return x + y
end
```

Definición mediante el operador function

El operador function se puede utilizar en una expresión para definir una función anónima:

```
function([listaParámetros])
  cuerpo
end
```

Cuando se define una función sin nombre, se define lo que se conoce como función anónima (*anonymous function*).

A continuación, se muestra cómo definir la operación anterior para la suma de dos números mediante el operador function:

```
local suma = function(x, y)
  return x + y
end
```

Parámetros

Toda función puede declarar cero, uno o más parámetros, donde un parámetro (*parameter*) no es más que una variable local de la función, cuyo valor se asigna en el momento de invocarse, o sea, su valor inicial se pasa en la llamada a la función. Se conoce como aridad (*arity*) al número de parámetros declarados en la definición de la función.

Todo parámetro especificado en la signatura debe tener un nombre para poder referenciarlo posteriormente en el cuerpo de la función. La sintaxis de los parámetros es la siguiente:

```
nombre
```

Así, por ejemplo, si deseamos realizar la resta de dos números, podemos definir una función como la siguiente:

```
function resta(x, y)
  return x - y
end
```

x e y son los parámetros, cuyos valores iniciales se asignarán en el momento de la llamada. He aquí un ejemplo de llamada a la función mediante la cual x se iniciará a 1 e y a 2:

```
resta(1, 2)  --x=1; y=2
```

Más adelante, presentaremos formalmente el concepto de invocación o llamada a función. Por ahora, observe que consiste en indicar el nombre de la función seguido de cero, una o más expresiones, los valores a asignar a los parámetros, delimitadas todas ellas por paréntesis y separadas por comas.

Parámetro de resto

El parámetro de resto (*rest parameter*) es un tipo especial de parámetro que recoge todos los valores pasados a la función que no tienen asignado un parámetro en la signatura de la función. Cada elemento representa un valor.

El parámetro de resto debe ser el último de la signatura y se declara mediante tres puntos (...).

Por ejemplo, si deseamos realizar la suma de dos o más números, podemos definir una función como la siguiente:

```
function suma(x, y, ...)
  local res = x + y
```



```
  for _, val in ipairs({...}) do res = res + val end
  return res
end
```
El parámetro resto recogerá los valores pasados en la invocación de la función a partir del tercero, inclusive, porque los dos primeros se asignarán a las variables x e y. Ejemplo:

```
suma(1, 2)        --x=1; y=2; ...={}
suma(1, 2,  3)    --x=1; y=2; ...={3}
suma(1, 2, 3, 4)  --x=1; y=2; ...={3,4}
```

Cuerpo de las funciones

El cuerpo (*body*) de una función es la secuencia o bloque de proposiciones que realiza la tarea asignada a la función. Toda función tiene un cuerpo, aunque no tenga asociada ninguna proposición.

Resultado de la función

En Lua, las funciones pueden devolver cero, uno o más valores. Hay que utilizar la sentencia return para indicar el valor o los valores a devolver. La sintaxis de esta sentencia es la siguiente:

```
return
return Expresión
return Expresión, Expresión, Expresión...
```
Cuando el motor de Lua se encuentra con la sentencia return, automáticamente finaliza la ejecución de la función y pasa, entonces, a la proposición que realizó la llamada. Si no se adjunta ninguna expresión, devolverá nil. Si por el contrario se especifica una expresión, el resultado obtenido de su evaluación será lo devuelto por la sentencia return.

Todos los ejemplos presentados hasta el momento devolvían algo. En el caso de la función de suma, el resultado de sumar los parámetros pasados a la función. Volvamos a ver uno de ellos:

```
function suma(x, y, ...)
  local res

  --(1) sumamos
  res = x + y

  for _, val in ipairs({...}) do
    res = res + val
  end

  --(2) devolvemos resultado
  return res
end
```
Al valor devuelto por la función se le conoce formalmente como valor de retorno (*return value*).

Invocación de funciones

Hasta el momento, hemos visto cómo definir las funciones, más concretamente, los objetos funciones. Pero las funciones no se ejecutan hasta que se solicita explícitamente. Cuando se definen lo que se hace es crear un objeto especial que contiene una secuencia de proposiciones. Nada más.

Cuando se desea ejecutar una función, se realiza lo que se conoce como llamada a función (*function call*) o invocación de función (*function invoke*), una expresión mediante la cual se solicita la ejecución del cuerpo de la función. Esta llamada o invocación se puede realizar como sigue:

```
función()
función argumento
función(argumento1, argumento2, argumento3...)
```
En Lua, el operador de llamada es obligatorio, salvo cuando se invoca con un único argumento. Aunque por convenio, se utiliza siempre el operador de llamada, es decir, los paréntesis.

Cuando una función llama a otra, la función que llama se conoce como llamadora (*caller*) y la invocada como llamada (*callee*).

Adaptación de los valores de retorno

La sentencia return puede devolver cero, uno o más valores. Cuando devuelve uno o más valores, el

valor devuelto se adapta a la expresión de invocación. Por ejemplo, si tenemos una asignación múltiple donde el resultado es el origen de la asignación de varias variables, su resultado se repartirá entre las variables. Ejemplo:

```
x = fn()          --a 'x' se le asigna el primer valor
x, y = fn()       --a 'x' se le asigna el primer valor; a 'y' el segundo
x, y, z = a, fn() --a 'x' se le asigna el valor de 'a'; a 'y' el 1º devuelto por la función;
                  --y a 'z' el segundo devuelto por la función
```

Si la función devuelve más valores que los necesitados por la asignación, los extras se perderán. Si la función devuelve menos valores, las variables que no reciban valor de la función se asignarán a nil.

También es posible crear una tabla con los valores devueltos por la función. Para ello, no hay más que utilizar la siguiente sintaxis:

```
{Expresión}
```

La expresión debe ser una invocación a función. Al delimitarse entre llaves, Lua entiende que se desea crear una tabla de tipo lista usando los valores devueltos como elementos. Veámoslo mediante unos ejemplos ilustrativos:

```
> function milista() return "uno", "dos", "tres" end
> x = milista()
> x
uno
> y = {milista()}
> y
table: 0x16b6550
>
```

Invocaciones protegidas

Una invocación protegida (*protected call*) es un tipo especial de invocación que asegura que la función no propagará ningún error en caso de producirse. Se realiza mediante la función pcall():

```
function pcall(función, arg1, arg2, arg3...)
```

Como primer argumento, se pasa el objeto de la función a invocar. Los restantes argumentos de la llamada se pasarán como argumentos a la función a invocar. pcall() devuelve dos valores:

- El primero indica si se ha propagada algún error: true, sí; false, no.
- El segundo representa el valor devuelto por la función.

Sobrecarga de funciones

La sobrecarga (*overload*) permite que dos o más funciones tengan el mismo nombre, pero siempre y cuando cada una de las implementaciones tenga una lista de parámetros distinta, es decir, tengan una signatura distinta. El tipo de retorno puede ser el mismo o distinto, pero de cara a la sobrecarga sólo se tiene en cuenta los tipos de los parámetros.

Lua *no* permite la sobrecarga de funciones al igual que JavaScript. En caso de ser necesaria, hay que implementar la función para que simule las distintas sobrecargas, atendiendo a los argumentos recibidos en cada invocación. Siendo muy útil en estos casos el parámetro de resto, así como la función type().

Variables y funciones locales

No olvide que cuando se desea definir una variable como local, hay que hacerlo usando la sentencia local. Si lo hace sin local, se creará como global, aun si se encuentra dentro de una función. Lo mismo ocurre con las funciones: si no las define con local, se crearán como globales.

