# Tipos de datos

Lua es un lenguaje de tipos dinámicos. Al igual que en otros lenguajes dinámicos, el tipo de una variable es inferido en función del valor asignado. En esta afirmación hay que tener en cuenta dos conceptos:

- Declaración
- Asignación

Cuando hablamos de lenguajes de tipos estáticos, como por ejemplo C, C++, C#, Java entre otros, realizamos las dos operaciones. Primero declaramos la variable, que tiene que ser obligatoriamente de un tipo y luego le asignamos un valor, que debe ser concordante con el tipo declarado.

En Lua y en otros lenguajes dinámicos, se hacen las dos operaciones durante la asignación. Esto quiere decir que cuando hacemos una asignación `a = "Valor"`, estamos declarando una variable `a`, a la cual le asignamos el valor `"Valor"`. A partir del valor asignado, el intérprete de Lua es capaz de 'adivinar' el tipo. En el caso de nuestro ejemplo, `"Valor"` es un literal de cadena y entonces la variable `a` será del tipo `string`. Dicho esto, si intentamos declarar una variable en Lua sin asignarle un valor, el intérprete nos dará un error.

---

# Tipos básicos de Lua

En Lua se definen ocho tipos básicos:

| | |
|---|---|
| **nil** | Representa un valor nulo o inválido |
| **boolean** | Representa un valor booleano: verdadero, falso |
| **number** | Representa un número real en coma flotante de doble precisión |
| **string** | Representa una cadena de caracteres |
| **function** | Referencia a una función declarada |
| **userdata** | Representa cualquier otro tipo de datos almacenados en la variable |
| **thread** | Referencia un hilo para entornos multi-hilo |

| | |
|---|---|
| **table** | Referencia a un array asociativo multidimensional |

Podemos conocer el tipo de una variable en todo momento usando la función `type()`:

```
print(type(11.6))               --> number
print(type("Hola Mundo"))       --> string
print(type(print))              --> function
print(type(false))              --> boolean
print(type(nil))                --> string
print(type({a = 1}))            --> table
```

---

# nil

El tipo `nil` indica que no hay valor definido. Si el valor es retornado para una variable, quiere decir que la variable no está definida. Si este valor es retornado por una función significa que no hay valor a devolver. Por ejemplo:

```
> print(a)
nil
```

En el ejemplo anterior se intenta imprimir el valor de una variable `a`, pero el valor devuelto es `nil` . En este caso significa que la variable no ha sido declarada y por lo tanto no tiene valor. Recuerda que toda variable que se le asigne el valor `nil` desaparece, porque para que exista, debe tener un valor asignado diferente de `nil`.

Cuando se asigna `nil` a un elemento de una tabla, al igual que en las variables, este elemento es anulado y por consiguiente, eliminado. Si se asigna `nil` a una variable que tiene una referencia a una tabla, la tabla entera será anulada y eliminada. Escribe el siguiente código en un archivo y ejecútalo con Lua para ver un ejemplo que te permita comprender mejor este concepto:

```
tabla1 = { elm1 = "val 1", elm2 = "val 2", elm3 = "val 3" }

for k, v in pairs(tabla1) do
    print(k .. " _ " .. v)
end

tabla1.elm2 = nil

for k, v in pairs(tabla1) do
    print(k .. " _ " .. v)
end
```

El resultado es el siguiente:

```
elm3 _ val 3
elm2 _ val 2
elm1 _ val 1
elm3 _ val 3
elm1 _ val 1
```

Podemos ver que las tres primeras lineas corresponden a los tres elementos de la tabla. Pero en las dos últimas, falta el elemento `elm2` que hemos eliminado asignándole `nil`.

---

# boolean

Los tipos boolean representan dos posibles valores: true o false. Puedes probar el siguiente código para entender mejor el concepto:

```
a = true
b = false

function PrintValue(value)
    if value then
        print("Verdadero")
    else
        print("Falso")
    end
end

PrintValue(a) --> a = true
PrintValue(b) --> b = false
```

El ejemplo produce la siguiente salida:

```
Verdadero
Falso
```

---

# number

En Lua existe un solo tipo numérico con el que se pueden representar varios formatos numéricos, por ejemplo:

- 5

- 5.1
- 0.6
- 3e+2
- 0.4e-1
- 3.423946103e+06

---

## string

Este tipo representa una cadena de caracteres. Esta cadena se puede delimitar usando comillas dobles (") o comillas simples (') indistintamente. Así por ejemplo son expresiones de cadenas válidas:

```
cadena1 = "Cadena con comillas dobles"
cadena2 = 'Cadena con comillas simples'
```

Cuando queremos asignar una cadena que contiene múltiples lineas, podemos usar dobles corchetes "[[]]" para englobar toda la cadena:

```
text = [[ Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vivamus ornare viverra augue, nec auctor sem pellentesque ac.
Nulla at varius quam.
Ut velit augue, ornare a sapien et, ultricies egestas mauris.
Nam augue nibh, vulputate ut sagittis eu, vulputate imperdiet dolor.  ]]
```

Si realizamos una operación matemática entre un tipo numérico y un tipo string, el intérprete de Lua intentará convertir la cadena de texto en un valor numérico y realizar la operación después. Si la cadena no puede ser convertida a un número el intérprete dará un error:

```
> print(8 + "2")
10
> print("8" + "2")
10
> print("8 + 2")
8 + 2
> print("3e2" * "3")
900
> print("numero" + 5)
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
stdin:1: in main chunk
[C]: in ?
```

Dos o más cadenas de texto se pueden concatenar usando dos puntos consecutivos (..) entre las cadenas o variables:

```
> a = "Conca"
> b = "tenación"
> print(a .. b)
Concatenación
> print("Conca" .. "tenación")
Concatenación
```

Se puede obtener la longitud de una cadena usando el operador almohadilla (#):

```
> cadena = "Longitud"
> longitud = #cadena
> print(longitud)
8
> print(#cadena)
8
> print(#"Longitud")
8
```

---

# table

El tipo table corresponde a un array asociativo, esto significa, que podemos usar números y cadenas para indizar los elementos. Del mismo modo, el array puede ser multinivel. Podemos inicializar una tabla vacía o añadiendo elementos:

```
tabla1 = {}


tabla2 = { "naranja", "manzana", "piña", "plátano" }
```

En una tabla, los índices no tienen que ser necesariamente del mismo tipo, pudiendo existir índices de tipo cadena y de tipo numérico:

```
tbl = {}
tbl["indice"] = "valor"


i = 5
tbl[i] = 10
tbl[i] = tbl[i] + 5
```

```
for k, v in pairs(tbl) do
    print(k .. " : " .. v)
end
```

Si ejecutamos el código anterior obtenemos el siguiente resultado:

```
$ lua TestTableIndices.lua
indice : valor
5 : 15
```

Las tablas en Lua son de índice 1, esto significa que por defecto, el primer elemento de una tabla cuando no se especifiquen índices explícitamente será 1. Esto marca una diferencia con multitud de lenguajes cuyos arrays son de índice 0. Veamos un ejemplo:

```
tabla = { "naranja", "manzana", "pera", "uva" }

for k, v in pairs(tabla) do
    print("Índice: " .. k .. ", Valor: " .. v)
end
```

Cuyo resultado es:

```
$ lua TestTableIndices2.lua
Indice: 1, Valor: naranja
Indice: 2, Valor: manzana
Indice: 3, Valor: pera
Indice: 4, Valor: uva
```

Las tablas en Lua son dinámicas por lo que no tienen un tamaño definido. Una tabla irá ajustando su tamaño en función del número de elementos que contenga automáticamente. Eso quiere decir, que es el intérprete de Lua quien se ocupa de gestionar la memoria necesaria para almacenar los elementos de la tabla.

---

# function

El tipo function almacena una referencia a una función. Esto significa que podemos asignar una función a una variable para usarla después:

```
function factorial(n)
    if n == 0 then
        return 1
```

```
    else
        return n * factorial(n - 1)
    end
end

print(factorial(4))
factorialVar = factorial
print(factorialVar(4))
```

Al ejecutar el código anterior obtenemos el siguiente resultado:

```
$ lua TestTipoFunction.lua
24
24
```

El tipo function también acepta las funciones anónimas pasadas como parámetros:

```
function Anonima(tbl, fun)
    for k, v in pairs(tbl) do
        print(fun(k, v))
    end
end

tabla = { clave1 = "valor1", clave2 = "valor2" }

Anonima(tabla, function(clave, valor)
    return clave .. " = " .. valor\nend)
```

El resultado del código anterior es el siguiente:

```
$ lua TestFunctionAnonima.lua
clave2 = valor2
clave1 = valor1
```

---

# thread

Las variables de tipo thread contienen referencias a hilos gestionados por el sistema operativo. Estos hilos permiten la realización de tareas en paralelo pero, pueden ser bastante difíciles de manejar. Como alternativa a los hilos, Lua maneja las llamadas corutinas, que permiten cierto paralelismo usando un solo hilo. Las corutinas son gestionadas directamente por el intérprete de Lua.

# userdata

Las variables de tipo userdata contienen referencias a datos arbitrarios de C de cualquier tipo. En este tipo de datos se suelen almacenar referencias a estructuras y punteros.

Revisión #5
Creado 23 mayo 2023 06:54:33 por Guillermo
Actualizado 31 julio 2023 12:02:47 por Guillermo