Funciones

Conceptos avanzados

Funciones

- Es posible definir funciones con número variable de argumentos

Estructura

- **function** nombre(...)
  Instrucciones
  end

Funciones

- ¿Cómo se accede a los parámetros?
- Mediante un array llamado **arg** que se define implícitamente
- Parámetros
  - Primer parámetro: arg[1]
  - Segundo parámetro: arg[2]
- ¿Cómo se cuantos parámetros hay?
- Mediante un campo en la tabla **arg** → n
  - Número de parámetros: arg.n

Ejemplo

```
function concatenar(...)
	message = ""
	for i = 1, arg.n do
		message = message..arg[i]
	end
	return message
end

print(concatenar("Hello", "World"))
```


Ejemplo

```
function concatenar(...)
	message = ""
	for i = 1, arg.n do
		message = message..arg[i]
	end
	return message
end

print(concatenar("Hello", " ", "World"))
```


Ejemplo

```
function concatenar(...)
	message = ""
	for i = 1, arg.n do
		message = message..arg[i]
	end
	return message
end

print(concatenar("Hello", " ", "World"" ", "my", " ", "name", " ",
"is", " ", "Diego"))
```

Funciones

- Es posible especificar parámetros fijos y el resto variable

Estructura

- **function** name([parametro]+, ...)
  Instrucciones
  end

Ejemplo

```
function concatenar(separador, ...)
	message = ""
	for i = 1, arg.n do
		message = message..arg[i]
		if i < arg.n then
			message = message..separador
		end
	end
	return message
end
print(concatenar(" ", "Hello", "World", "my", "name", "is", "Diego"))
```


Arrays como parámetros

- Para funciones de número variable de parámetros
- Mucho más cómodo para listas largas de parámetros

Unpack

- Función que devuelve los elementos de un array
- Puede usarse para llamadas a funciones
- Puede usarse para asignaciones

Ejemplo

```
function concatenar(separador, ...)
	message = ""
	for i = 1, arg.n do
		message = message..arg[i]
		if i < arg.n then
			message = message..separador
		end
	end
	return message
end

palabras = {"Hello", "World", "my", "name", "is", "Diego"}
print(concatenar(" ", unpack(palabras))
```


Tablas

La estructura de datos en lua

Qué son?

- Son la única estructura de datos en lua
- En otros lenguajes hay muchas
  - Structs
  - Clases
  - Listas
  - Diccionarios
  - Tuplas
  - Etc.
- Las hemos usado para crear arrays
- Pero son mucho más generales y flexibles

Qué son?

- Arrays asociativos
  - Arrays que pueden indexarse con cualquier tipo de dato
  - Similar a los diccionarios de python

Estructura

- variable = {}
- Variable[<indice>] = valor
- Índice puede ser un valor de cualquier tipo
  - Incluso otra tabla

Ejemplos

```
vida_enemigos = {}
vida_enemigos["goblin"] = 25
vida_enemigos["grunt"] = 200
vida_enemigos["hobbit"] = 5
```

Índice	valor
goblin	25
grunt	200
hobbit	5

```
datos = {}
datos["dragon"] = 300
datos[1] = "mage"
datos[2] = "kobold"
datos[3] = "dragon"
datos["mage"] = 4
```

Índice	valor
dragon	300
1	mage
2	kobold
3	dragon
mage	4

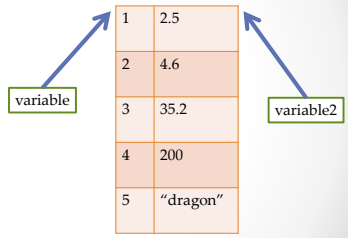
Cómo se usan?

- No tienen tamaño fijo
- No hay que especificar el tamaño
  - Aumentan dinámicamente al añadir datos
  - Se reducen automáticamente al eliminar datos
- Se gestionan por referencia
  - Si igualamos una tabla a otra no se copian los contenidos
  - La nueva variable hará referencia al mismo objeto
  - Cambios en la nueva variable se verán usando la antigua
- La variable es sólo una especie de puntero

Ejemplo

```
variable = {}
variable[1] = 2.5
variable[2] = 4.6
variable[3] = 35.2
variable[4] = 200
variable[5] = "dragon"

variable2 = variable
```



Valores

- Igual que las variables normales
- Si no está asignado ningún valor → su valor es **nil**

```
creature = {}
creature["health"] = 300
print(creature["ammo"])
```

- El valor puede ser de cualquier tipo

```
creature["ammo"] = 22.5
creature["name"] = "Zulrog"
creature["alive"] = true
```


Elementos

Añadir

- Asignar un valor

```
creature = {}
creature["health"] = 300
```

Eliminar

- Asignar **nil**

```
creature = {}
print(creature["ammo"])
creature["ammo"] = 30.2
creature["ammo"] = nil
print(creature["ammo"])
```

Arrays generales

- Un array es simplemente una tabla cuyos índices son numéricos
- Al estar basados en tablas son generales

```
array = {}
array[-1] = 5
array[0] = 10
array[1] = 15
```


Arrays sparse

- Arrays que no tienen un valor para cada índice

```
level_items = {}
level_items[2] = "potion"
level_items[5] = "chest"
```


Arrays generales

- Ciertas operaciones no son tan generales
- Trabajan sobre arrays estándar
  - Empiezan en índice 1
  - No admiten índices vacíos
    - Valores nil a mitad del array
- Operaciones
  - Longitud

```
level_items = {}
level_items[2] = "gold"
level_items[5] = "chest"
print(#level_items)
```

```
inventory = {}
inventory[1] = "key"
inventory[2] = "bear chest"
inventory[3] = "hat of haste"
print(#inventory)
inventory[2] = nil
print(#inventory)
```

Resumen

- Una tabla es una lista de asociaciones clave – valor
- La clave puede ser de cualquier tipo
- El valor puede ser de cualquier tipo
- Para tener un array
  - Las claves deben ser numéricas
  - Empezar en 1
  - Todos los elementos con claves contiguas (números consecutivos)

Acceso (variante)

- Otro tipo de acceso a los valores de una tabla
- Más cómodo en ciertos casos
  - Sobre todo si se viene de otros lenguajes como C o C++
- Válido para claves de tipo cadena
  - No sirve para acceder a los valores indexados por un número
- No se permiten espacios en la clave
- Lo llamaremos acceso tipo struct

Estructura

- array.<clave>

Ejemplo

```
creatures = {}
creatures["ammo"] = 19
creatures["health"] = 400
creatures["name"] = "Thathung"
creatures["class"] = "grunt"
print(creatures["ammo"])
print(creatures.ammo)
```

```
creatures = {}
creatures.ammo = 19
creatures.health = 400
creatures.name = "Thathung"
creatures.class = "grunt"
print(creatures["ammo"])
print(creatures.ammo)
```

Tablas anidadas

- Los valores pueden ser de cualquier tipo
- Esto incluye otras tablas

```
creature = {}
creature["name"] = "Zack Zero"
creature["items"] = {"fire suit", "scroll", "key"}

mission = {}
mission["goal"] = "rescue Marlene"
mission["score"] = 4000
mission["secrets"] = 3
creature["mission"] = mission
```


Constructores

- Hay varias formas de construir una tabla
- El resultado final es independiente de cómo se construyo la tabla
- Es cuestión de comodidad al escribir

Constructores: Array

Estructura

- array = {valor1, valor2, valor3, ...}
- El primer valor irá al índice 1
- El resto irán a índices consecutivos

```
creatures = {"goblin", "troll", "grunt", "golem", "human"}
print(creatures[1])
print(creatures[3])
print(creatures[7])
```

Constructores: Tabla

- Estructura
- tabla = { [clave] = valor, [clave] = valor, ...}

```
creature = {
      ["x"] = 23.45,
      ["y"] = 200.12,
      ["name"] = "Zack Zero",
}
```

```
traps = {
      [10] = "explosive",
      [12] = "fireball",
      [25] = "hole",
      ["level"] = "fire & ice",
}
```


Constructores: Struct

Estructura

- variable = {clave = valor, clave = valor, ...}
- Las claves se añaden como si fueran claves de tipo cadena
- No se permiten espacios en las claves

```
creature = { x = 25, y = 50, name = "Zulrog"}
print(creature.x)
print(creature.name)
print(creature["x"])
```


Constructores: anidados

- Dentro de un constructor podemos tener otro

```
creature = {
    name = "Zack Zero",
    items = {"fire suit", "scroll", "key"},
    mission = {
       goal = "rescue Marlene",
      score = 4000,
      secrets = 3
    }
}
```

Funciones como valores

- Las funciones son un tipo de dato
- Igual que los números, las cadenas o las tablas
- El nombre de la función es realmente una variable
- Podemos asignar valores de función

```
function sayHello()
      print("Hello")
end
diHola = sayHello
diHola()
```

```
function sayHello()
      print("Hello")
end
diHola = sayHello()
diHola()
```


Funciones en tablas

- También pueden añadirse como valores en una tabla

```
function gruntSpeak()
      print("For the horde!!!")
end

creature = {}
creature.speak = gruntSpeak
creature.class = "grunt"
```


Funciones en tablas

```
function gruntSpeak()
      print("For the horde!!!")
end
function knightSpeak()
      print("Yes, milord")
end
```

```
grunt = {}
grunt.speak = gruntSpeak
grunt.class = "grunt"

knight = {}
knight.speak = knightSpeak()
knight.class = "knight"

creatures = {grunt, knight}
for i = 1, #creatures do
      creatures[i].speak()
```

Funciones en tablas

```
function gruntSpeak()
      print("For the horde!!!")
end
function knightSpeak()
      print("Yes, milord")
end
```

```
creatures = {
      { speak = gruntSpeak,
        class = "grunt"
      },
      { speak = knightSpeak,
        class = "knight"
      }
}
for i = 1, #creatures do
      creatures[i].speak()
```


Iteración

- En muchas ocasiones hay que hacer una operación en cada elemento de una tabla
- Para ello hay que recorrerse la tabla
- Ya hemos visto como recorrer una tabla que funciona como array

```
creatures = {"mage", "goblin", "orc", "kobold"}
for i = 1, #creatures do
      print(creatures[i])
end
```


Iteración tablas

- ¿Y si quiero recorrer una tabla que no es un array?
- For genérico
- Iteradores

For genérico

Estructura

- **for** var1, var2, … **in** iterador **do**
  Instrucciones
  end

Iteradores (Array)

ipairs

- Función para recorrer los elementos de un array

Estructura

- **for** i, v **in** ipairs(array) **do**
  Instrucciones
  end
- i → índice del elemento
- v → valor del elemento

Ejemplo

```
creatures = {"mage", "goblin", "orc", "kobold"}

for i = 1, #creatures do
      print(creatures[i])
end

for i, v in ipairs(creatures) do
      print(v)
```

Ejemplo

```
creatures = {
      { speak = gruntSpeak,
        class = "grunt"
      },
      { speak = knightSpeak,
        class = "knight"
      }
}

for i, v in ipairs(creatures) do
      v.speak()
```


Iteradores (Tablas)

pairs

- Función para recorrer los elementos de una tabla

Estructura

- **for** k, v **in** pairs(tabla) **do**
  Instrucciones
  end
- k → clave del elemento
- v → valor del elemento

Ejemplo

```
creature = {
      x = 30.0
      y = 200.4
      name = "Zack Zero"
      armor = "fire"
}
for k, v in pairs(creature) do
      print(k, v)
```

Ejemplo

```
function printTable(tabla)
      for k, v in pairs(tabla) do
            print(k, v)
      end
end
creature = {
      x = 30.0,
      y = 200.4,
      name = "Zack Zero",
      armor = "fire",
}
printTable(creatures)
```


Ejemplo

```
creature = {
    name = "Zack Zero",
    items = {"fire suit", "scroll", "key"},
    mission = {
        goal = "rescue Marlene",
       score = 4000,
       secrets = 3
    }
}
printTable(creature)
```


Ejemplo

```
creature = {
    name = "Zack Zero",
    items = {"fire suit", "scroll",
"key"},
    mission = {
        goal = "rescue Marlene",
            score = 4000,
            secrets = 3
    }
}
```

```
function printTable(tabla, tab)
      for k, v in pairs(tabla) do
            if type(v) == "table" then
                  print(tab..k.." is table")
                  printTable(v, tab..."    ")
            else
                  print(tab..k.." is "..v)
            end
      end
end

printTable(creature, "")
```

Precauciones

- Malos usos de la flexibilidad
  - Complicar el código
  - Mantenibilidad futura
  - Cuidado con las estructuras "creativas" u "originales"
- Las claves de una tabla son referencias
  - Indexar con una tabla requiere exactamente la misma tabla
    - No es suficiente que tengan los mismos datos

```
relation = { [{name = "Zulrog"}] = "Final Boss"}
print(relation[{name = "Zulrog"}])
```


Precauciones

```
zulrog = { name = "Zulrog"}
relation = { [zulrog] = "Final Boss"}
print(relation[zulrog])
```

```
zulrog = { name = "Zulrog"}
relation = { [zulrog] = "FinalBoss"}
other_zulrog = { name = "Zulrog"}
print(relation[other_zulrog])
```

```
zulrog = { name = "Zulrog"}
relation = { [zulrog] = "FinalBoss"}
zulrog = { name = "Zulrog"}
print(relation[zulrog])
```