Programming in Lua – Data Structures

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiomlua
Tables for everything

• Unless you resort to C code, tables are the only way to structure data in Lua.

• They can represent arrays, sets, records, objects, and other data structures efficiently, with a nice syntax.

• The basic operations that a table supports are construction ({}), to make a new table, and indexing ([ ]), to read/write values.

> tab = {}  -- make a new table assign to tab
> tab["x"] = 5   -- write 5 to “x” field
> print(tab["x"])  -- read value of “x” field and print it
5

• Tables are a mutable reference type, so they have the same aliasing issues as C pointers and Java arrays and objects.

> alias = tab
> alias["x"] = "changed"
> print(tab["x"])  
changed
Arrays

- A Lua array is a table with values assigned to sequential integer keys, starting with 1

```lua
local a = {}
for i = 1, 6 do
  a[i] = math.random(10)
end
```

- You can initialize an array using a table constructor with a list of expressions inside it

```lua
-- an array like the previous one
local a = { math.random(10), math.random(10), math.random(10),
            math.random(10), math.random(10), math.random(10) }
```

- An array cannot have holes: none of the values can be nil. But you can fill the array in any order you want, as long as you plug all the holes before using the array
Length

- The length operator (#) gives the number of elements in an array.

- You can use the length and a for loop to iterate over an array:

```lua
local a = { math.random(10), math.random(10), math.random(10),
            math.random(10), math.random(10), math.random(10) }
for i = 1, #a do
    print(a[i])
end
```

- The length operator is also useful for adding elements to the end of an array, and removing the last element:

```lua
a[#a] = nil -- remove the last element
a[#a + 1] = math.random(10) -- add a new element to the end
```
• Two functions in the table module can insert and remove elements in any position of the array (shifting the other elements to make space or plug the hole, respectively):

```lua
> a = { 1, 2, 3, 4, 5 }
> table.insert(a, 3, 10) -- insert 10 in position 3
> print_array(a)
{ 1, 2, 10, 3, 4, 5 }
> table.remove(a, 4) -- remove fourth element
> print_array(a)
{ 1, 2, 10, 4, 5 }
```

• The function `table.sort` sorts an array (using an efficient sorting algorithm):

```lua
> a = { "Python", "Lua", "C", "JavaScript", "Java", "Lisp" }
> table.sort(a)
> print_array(a)
{ C, Java, JavaScript, Lisp, Lua, Python }
```
Concatenation

- The `table.concat` function concatenates an array of strings using an optional separator:

```lua
function print_array(a)
    print("{" .. table.concat(a, ",", ) .. "}")
end
```

- If we do not give a separator then `concat` uses ""

- A common idiom is to use an array of strings as a buffer, using `table.concat` when we need the contents of the buffer as a single string
Iteration with ipairs

Another way to iterate over an array is to use the *generic* for loop and the ipairs built-in function:

```lua
local a = { 1, 3, 5, 7, 9 }
local sum = 0
for i, x in ipairs(a) do
    print("adding element ", i)
    sum = sum + x
end
print("the sum is ", sum)
```

This loop has two control variables, the first gets the indices, the second gets the elements.

Usually we are only interested in the elements and not the index, so it is common to use `_` as the name of the control variable for the indices.
Matrices

• One way to represent multi-dimensional arrays is with “jagged arrays”, as in Java, where you have an array of arrays for two dimensions, an array of arrays of arrays for three, etc.

```lua
local mt = {}
for i = 1, 3 do
    mt[i] = {}
    for j = 1, 5 do
        mt[i][j] = 0
    end
end
```

• A more efficient way is to compose the indices using multiplication, as C does for you:

```lua
local mt = {}
for i = 1, 3 do
    for j = 1, 5 do
        mt[(i-1)*5+j] = 0
    end
end
```
Records

• A Lua record is a table with string keys, where they keys are valid Lua identifiers; you can initialize record fields in the table constructor by passing key/value pairs:

  ```lua
  point1 = { x = 10, y = 20 }
  point2 = { x = 50, y = 5 }
  line   = { from = point1, to = point2, color = "blue" }
  ```

• You can read and write record fields with the . operator:

  ```lua
  line.color = "red"  -- same as line["color"] = "red"
  print(line.from.x, line["color"])
  ```

• A table can be both a record and an array, and you can initialize both parts in a single table constructor
Sets

- A nice way to represent sets in Lua is with a table where the *keys* are the elements of the set, and the values are *true*.

- This way, you can test membership in the set and add or remove elements to the set by indexing.

- You can initialize the set with a third table constructor syntax:

  ```lua
  local set = { [math.random(10)] = true, [math.random(10)] = true,
               [math.random(10)] = true, [math.random(10)] = true }
  ```

- If you replace true by a number and use it as a counter you have a *multiset*, or a set that can have more than one copy of each element.
Iteration with pairs

- A for loop using the pairs built-in function iterates over all keys and values of a table:

```lua
local tab = { x = 5, y = 3, 10, "foo" }
for k, v in pairs(tab) do
    print(tostring(k) .. " = " .. tostring(v))
end
```

- The first control variable gets the keys, the second gets the values

- pairs does not guarantee an order of iteration, even among numeric keys; use ipairs if you want to iterate over an array

- But pairs is great to iterate over a set, use _ as the name of the control variable for the values, as you will not need it (unless you have a multiset, of course!)
Quiz

What will be the output of the following program?

```python
sunday = "monday"; monday = "sunday"
t = { sunday = "monday", [sunday] = monday }
print(t.sunday, t[sunday], t[t.sunday])
```

```
t("sunday") t["monday"] t["monday"]
t["sunday"] = "monday"
t(sunday) = monday  →  t["monday"] = "sunday"
{ sunday = "monday", monday = "sunday" }
```