LUA --> Means moon in portuguese

Lua is a case-sensitive language

**Data types**

1. number
2. bool
3. string
4. functions
5. Tables
6. user datatype
7. Thread

Io.read() 🡪 Reading from keyboard

Io.write() 🡪 Writing data to screen

**K*eywords*:**

and break do else elseif end

false for function goto if in

local nil not or repeat return

then true until while

**C-like escape sequences:**

1. '\a' (bell),
2. '\b' (backspace),
3. '\f' (form feed),
4. '\n' (newline),
5. '\r' (carriage return),
6. '\t' (horizontal tab),
7. '\v' (vertical tab),
8. '\\' (backslash),
9. '\"' (quotation mark [double quote]),
10. ‘\'' (apostrophe [single quote])

**Variables:**

3 types :

* Global variables
* Local variables
* Table fields

Any variable name is assumed to be global unless explicitly declared as a local. Local variables are lexically scoped: local variables can be freely accessed by functions defined inside their scope.

Note: If a variable has not given the data then it is **nil** data

**Control Structures:**

The control structures **if**, **while**, and **repeat** have the usual meaning and familiar syntax:

**While:** **while** exp **do** block **end**

**Repeat:** **repeat** block **until** exp

**If:** **if** exp **then** block {**elseif** exp **then** block} [**else** block] **end**

Both **false** and **nil** are considered false and All values different from **nil** and **false** are considered true

3.3.5 – For Statement

The **for** statement has two forms: one numerical and one generic.

The numerical **for** loop repeats a block of code while a control variable runs through an arithmetic progression. It has the following syntax:

stat ::= **for** Name ‘**=**’ exp ‘**,**’ exp [‘**,**’ exp] **do** block **end**

The *block* is repeated for *name* starting at the value of the first *exp*, until it passes the second *exp* by steps of the third *exp*. More precisely, a **for** statement like

for v = *e1*, *e2*, *e3* do *block* end

is equivalent to the code:

do

local *var*, *limit*, *step* = tonumber(*e1*), tonumber(*e2*), tonumber(*e3*)

if not (*var* and *limit* and *step*) then error() end

*var* = *var* - *step*

while true do

*var* = *var* + *step*

if (*step* >= 0 and *var* > *limit*) or (*step* < 0 and *var* < *limit*) then

break

end

local v = *var*

*block*

end

end

Note the following:

* All three control expressions are evaluated only once, before the loop starts. They must all result in numbers.
* *var*, *limit*, and *step* are invisible variables. The names shown here are for explanatory purposes only.
* If the third expression (the step) is absent, then a step of 1 is used.
* You can use **break** and **goto** to exit a **for** loop.
* The loop variable v is local to the loop body. If you need its value after the loop, assign it to another variable before exiting the loop.

The generic **for** statement works over functions, called *iterators*. On each iteration, the iterator function is called to produce a new value, stopping when this new value is **nil**. The generic **for** loop has the following syntax:

stat ::= **for** namelist **in** explist **do** block **end**

namelist ::= Name {‘**,**’ Name}

A **for** statement like

for *var\_1*, ···, *var\_n* in *explist* do *block* end

is equivalent to the code:

do

local *f*, *s*, *var* = *explist*

while true do

local *var\_1*, ···, *var\_n* = *f*(*s*, *var*)

if *var\_1* == nil then break end

*var* = *var\_1*

*block*

end

end

Note the following:

* *explist* is evaluated only once. Its results are an *iterator* function, a *state*, and an initial value for the first *iterator variable*.
* *f*, *s*, and *var* are invisible variables. The names are here for explanatory purposes only.
* You can use **break** to exit a **for** loop.
* The loop variables *var\_i* are local to the loop; you cannot use their values after the **for** ends. If you need these values, then assign them to other variables before breaking or exiting the loop.

Arithmetic Operators

Lua supports the following arithmetic operators:

* **+:**addition
* **-:**subtraction
* **\*:**multiplication
* **/:**float division
* **//:**floor division
* **%:**modulo
* **^:**exponentiation
* **-:**unary minus

Bitwise Operators

Lua supports the following bitwise operators:

* **&:**bitwise AND
* **|:**bitwise OR
* **~:**bitwise exclusive OR
* **>>:**right shift
* **<<:**left shift
* **~:**unary bitwise NOT

Bitwise operators always convert float operands to integers.

string concatenation accepts numbers as arguments, besides strings.

Relational Operators

Lua supports the following relational operators:

* **==:**equality
* **~=:**inequality
* **<:**less than
* **>:**greater than
* **<=:**less or equal
* **>=:**greater or equal

These operators always result in **false** or **true**.

### Logical Operators

The logical operators in Lua are **and**, **or**, and **not**. Like the control structures (see [§3.3.4](http://www.lua.org/manual/5.3/manual.html#3.3.4)), all logical operators consider both **false** and **nil** as false and anything else as true.

The negation operator **not** always returns **false** or **true**. The conjunction operator **and** returns its first argument if this value is **false** or **nil**; otherwise, **and** returns its second argument. The disjunction operator **or** returns its first argument if this value is different from **nil** and **false**; otherwise, **or** returns its second argument. Both **and** and **or** use short-circuit evaluation; that is, the second operand is evaluated only if necessary. Here are some examples:

10 or 20 --> 10

10 or error() --> 10

nil or "a" --> "a"

nil and 10 --> nil

false and error() --> false

false and nil --> false

false or nil --> nil

10 and 20 --> 20

(In this manual, --> indicates the result of the preceding expression.)

### 3.4.8 – Precedence

Operator precedence in Lua follows the table below, from lower to higher priority:

* + or
  + and
  + < > <= >= ~= ==
  + |
  + ~
  + &
  + << >>
  + ..
  + + -
  + \* / // %
  + unary operators (not # - ~)
  + ^

As usual, you can use parentheses to change the precedences of an expression. The concatenation ('..') and exponentiation ('^') operators are right associative. All other binary operators are left associative.

Table:

Eg:

Simple table:

* Tab={1,2,3,”Sri”}
* Print(Tab)

Nested table 1:

* tab={1,222,3,4,t={44,55,66}}
* print(tab['t'][1])

Nested table 2:

* tab={1,222,3,4,{44,55,66}}
* print(tab[5][1])

**Modifying:**

* tab={1,222,3,4,{44,55,66}}
* tab[3]=4444

**Functions:**

A small chunk of code of program to perform some sopecific task

**Syntax:**

*function name(arguments)*

*code block*

*return*

*end*

*Note: In lua we can return multiple values.*

The statement

function f () body end

translates to

f = function () body end

The statement

function t.a.b.c.f () body end

translates to

t.a.b.c.f = function () body end

The statement

local function f () body end

translates to

local f; f = function () body end

not to

local f = function () body end

(This only makes a difference when the body of the function contains references to f.)

*Eg:*

-- Define function with multiple parameters and multiple return values.

function myFirstLuaFunctionWithMultipleReturnValues(a,b,c)

return a,b,c,"My first lua function with multiple return values", 1, true

end

a,b,c,d,e,f = myFirstLuaFunctionWithMultipleReturnValues(1,2,"three")

print(a,b,c,d,e,f)

-------- Output ------

1 2 three My first lua function with multiple return values 1 true