

LUA TYPES

Note Title

11/18/2010

→ Lua has 8 basic types:

a) nil

e) userdata

b) boolean

f) function

c) number

g) thread

d) string

h) table

→ The type function returns the type name for a variable

NOTE: type(type(20)) → string

OPERATOR PRECEDENCE

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

^
not - (unary)
* /
+ -
..
< > <= >= ~= ==
and
or

All binary operators are left associative, except for `^` (exponentiation) and `..` (concatenation), which are right associative. Therefore, the following expressions on the left are equivalent to those on the right:

$a+i < b/2+1 \quad \leftrightarrow \quad (a+i) < ((b/2)+1)$
 $5+x^2*8 \quad \leftrightarrow \quad 5+((x^2)*8)$
 $a < y \text{ and } y \leq z \quad \leftrightarrow \quad (a < y) \text{ and } (y \leq z)$
 $-x^2 \quad \leftrightarrow \quad -(x^2)$
 $x^y^z \quad \leftrightarrow \quad x^{(y^z)}$

When in doubt, always use explicit parentheses.

→ If you try to compare 2 different types using relational operators or 2 types having different metatable operators then lua raises an error.

→ == (equality) never raises an error. With 2 different types it always returns false without even calling any metamethods (if any)

→ and or let $x = \text{nil}$ & $n = 2$

print (x or 1) → 1
print (1 or x) → 1
print (x or x) → nil
print (1 or n) → 1
print (n or 1) → 2
print (x and 1) → nil
print (1 and x) → nil
print (1 and n) → 2
print (n and 1) → 1

Thus and/or returns more info than true/false when true it also returns the value of a particular parameter.

For 'and' it returns the value of the last parameter
for 'or' it returns the value of the 1st parameter

print (1 and n and 3) → 3
print (3 or n or 1) → 3

LOCAL VARIABLES

Besides global variables, Lua supports local variables. We create local variables with the local statement:

```
j = 10    -- global variable
local i = 1 -- local variable
```

Unlike global variables, local variables have their scope limited to the block where they are declared. A block is the body of a control structure, the body of a function, or a chunk (the file or string with the code where the variable is declared).

```
x = 10
local i = 1    -- local to the chunk
```

```
while i <= x do
  local x = i*2 -- local to the while body
  print(x)     --> 2, 4, 6, 8, ...
  i = i + 1
end
```

— NOTE: local variable access is faster

— Use local variables as normal variables inside functions etc. i.e. declare everything to local in Lua Programs

— Local variable declarations can also have multiple assignments like normal statements :

```
local a, b = 1, 10
if a < b then
  print(a) --> 1
  local a -- '= nil' is implicit
  print(a) --> nil
end -- ends the block started at 'then'
print(a,b) --> 1 10
```

CONTROL STRUCTURES

Generic FOR :

```
for var_1, ..., var_n in explist do
```

```
  block
end
```



```
do
```

```
  local _f, _s, _var = explist
  while true do
```

```
    local var_1, ..., var_n = _f(_s, _var)
```

```
    _var = var_1
```

```
    if _var == nil then break end
```

```
    BLOCK
```

```
  end
```

```
end
```

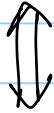
NUMERIC FOR

for var = exp1, exp2, exp3 do

↗ step increment (optional)

Block

end



do

local _exp1, _exp2, _exp3 = exp1, exp2, exp3

local step = _exp3 or 1

local var = _exp1

while var <= _exp2 do

Block

var = var + step

end

end.

FUNCTIONS

→ Variable number of arguments :

```
function testm(...)  
  print (arg.n)
```

→ ellipses to indicate variable number of arguments

-- print the total number of arguments passed

```
  for i,v in ipairs(arg) do  
    print (i,v)  
  end
```

(All arguments are in the table arg & it has the field 'n' containing number of arguments)

end

→ functions are first class values with lexical scoping

```
∴ function foo(x) return x+1 end  
    |||
```

↓
functions can access local variables of the functions enclosing them.

```
foo = function(x) return (x+1) end
```

TABLES

Table Constructor : $\{ \}$
ex : $a = \{ \}$

→ If a table is assigned another table as a metatable then that metatable is looked up to figure out the behaviour of using the tables with operators etc

ex $b = \{ \}$

Ex $\text{setmetatable}(a, b)$ -- b is set as metatable of a

Adding **special** key value pairs in the metatable b will decide how access & operators on a behave:

INITIALIZATION EXAMPLES

$b = \{ ["hello"] = 3, "good", [4] = "new", test = 8 \}$

↓ index = 1 index = 4 index = "test"

METATABLES

Field names :	-- add	→	+	(addition)
Arithmetic	-- mul	→	*	(multiplication)
	-- sub	→	-	(subtraction)
	-- div	→	/	(division)
	-- unum	→	-	(negation)
	-- pow	→	^	(exponentiation)
String Relational	-- concat	→	..	(concatenation)
	-- eq	→	==	(equality)
	-- lt	→	<	(less than)
	-- le	→	<=	(less or equal)
Lock from edit (metatable)	-- metatable	→	when slot locks the metatable.	
	-- index	→	To point to function or	

Example

```
b.--index = function (table, key)
  Do something
end
```

→ passes the table and key to the function

table to lookup to
find index before
returning nil.

```
b.--index = {x=1, y=2}
```

↳ A default table

--newindex → To contain a function
to call for making

Example: (Same as --index)

```
b.--newindex = function (table, key)
  Do something
end
```

an assignment in the
table to an index that
does not exist. It can also

NOTE: If --newindex metamethod is there
the table is not touched by Lua

contain a table in which
case the assignment is
made to that table

```
b.--newindex = {}
```

↳ Empty table to catch everything

NOTE: Lua code cannot change metatables of lua types
userdata, it can however read the metatable &
change its key value pairs.

