Lua Reference Sheet (Author: Luc Talatinian)

Key features

- Dynamic typing
- All types are first-class
- Versatility through table data
- Define table ops through metatables
- Threading w/ coroutines (non-preemptive)

Types

`nil`	different from everything else (except itself), acts as logical `false`
`NaN`	result of undefined ops (ie. `1/0`)
`boolean`	literal `true` or `false`
`number`	numeric values, subtypes are `float` and `int`
`string`	enclosed in single/double quotes, strings are immutable
`thread`	objects created using `coroutine.[]` functions
`userdata`	Memory passed from a C program
`table`	all-purpose data structure, uses key/value storage (see table section)
`type(val)`	returns type of given `val` (as string)

Core syntax

variables	`[local] var [= initial]`

functions	`function f(params)`
	`[statements]`
	`end`

array (table)	`arr = {val1, ..., valn}` index from `arr[1]` to `arr[n]`
math ops	`+ - * /`
modulus	`a % b`
concat	`str1 .. str2`
length	`#val` (strings, tables)
stdout	`print(value)`

Table concepts

- Key-value data storage
- Heterogeneous keys & values (can have any index/value type except `NaN` & `nil`)
- Can use to represent any data struct (ie. stack, queue, custom class)
- Metatables: table of functions for other table ops, can set manually for custom table structures

Table usage

Empty table	`{}`

non-array decl	`tab = {`
	`key1 = val1, ...`
	`}`

new value	any key type: `tab[key] = val` `tab.key = val` (strings)
delete value	`tab[key] = nil`
Metatables	keyed with `__[op]` If operation is encountered with unexpected operands, Lua will use op in metatable (think operator overloading)
set mt	`setmetatable(tab, mt)`

iteration	`for k,v in ipairs(t) do`
	`[iterate thru key&val]`
	`end`

Some useful metamethods

math	`__add, __sub, __mul, __div`
concatenation	`__concat`
indexing	`__index`
equality	`__eq`
less than	`__lt`
less/equal	`__le`
length	`__len`
bitwise	`__band, __bor, __bxor, __bnot`

Coroutine basics

- "collaborative multithreading"
- Not classical threads: no innate coroutine scheduling (round-robin, etc.)
- Use `coroutine` package (table of functions) to create and manipulate coroutines
- Thread management (scheduling) is left to user definition
- States: `suspended`, `running`, `dead`

Coroutine usage

new	`coroutine.create(func)` creates coroutine with func to execute does NOT auto-start
start	`coroutine.resume(co)` starts or resumes coroutine
pause	`coroutine.yield()` running -> suspended (call in coroutine func)
get state	`coroutine.status(co)` returns status of given routine (as string)
params	`resume()` returns any args given to `yield()`

Lua as an extension: core Lua in vim

`:lua [code]`	execute code in Lua
`:luafile [f]`	runs script in lua code file
`vim.command(c)`	execute ex command `c` in vim
`vim.window().line`	get line # of cursor in buffer
`vim.buffer()`	returns current editing buffer `b` (see below)
`b[n]`	access/modify/delete nth line of buffer
`b.newline(n)`	insert newline in buffer (end of buffer if no `n` given)