Lua crash course

Following crash course is an extremely condensed extract from the official Lua reference manual. For more details refer to the free Lua Quick Reference e-book, to the Lua website and to the numerous examples contained in the demo scenes.

Lexical conventions

- Lua is a case sensitive language. "and", "And" or "AND" are not the same.
- Following are Lua keywords: `and break do else elseif end false for function if in local nil not or repeat return then true until while`
- Following strings denote other tokens: `+ - * / % ^ # == ~= <= >= < > = ( ) { } [ ] ; : , . .. ...`
- Literal strings can be delimited by matching single or double quotes (e.g. `'hello'` or `"hello"`)
- A comment starts with a double hyphen (--) anywhere outside of a string. e.g.:

```
a=4 -- variable a is now 4!
```

Types and values

- Lua is a dynamically typed language which means that variables do not have types; only values do.
- There are 8 basic types in Lua:
  - `nil` type of the value nil whose main property is to be different from any other value. It usually represents the absence of a useful value
  - `bool` values false and true (both nil and false make a condition false; any other value makes it true)
  - `number` both integer and floating-point numbers (has internally two distinct representations: long integer and double)
  - `string` arrays of characters (strings may contain any 8-bit character, including embedded zeros)
  - `function` Lua functions
  - `userdata` can hold arbitrary C data (corresponds to a block of raw memory)
  - `thread` independent threads of execution used to implement coroutines
  - `table` arrays that can hold values of any type except nil

Variables

- There are 3 kinds of variables: global variables, local variables and table fields. Any variable is assumed to be global unless explicitly declared as local
- Before the first assignment to a variable, its value is nil
- Square brackets are used to index a table (e.g. `value=table[x]`). The first value in a table is at position 1 (and not 0 as for C arrays)

Statements

- Relational operators (always result in **false** or **true**)
  - `==` equality
  - `~=` negation of equality
  - `<` smaller than
  - `>` bigger than
  - `<=` smaller or equal than
  - `>=` bigger or equal than
- Lua allows multiple assignments. The syntax for assignments defines a list of variables on the left side and a list of expressions on the right side. The elements in both lists are separated by commas:

```
x,y,z = myTable[1],myTable[2],myTable[3]
```

- **If** control structure (by example):

```
if value1==value2 then
    print('value1 and value2 are same!')
end
```

- **For** control structure (by example):

```
for i=1,4,1 do -- count from 1 to 4 with increments of 1
    print(i)
end
```

- **While** control structure (by example):

```
i=0
while i~=4 do
    i=i+1
end
```

- **Repeat** control structure (by example):

```
i=0
repeat
    i=i+1
until i==4
```

- **Table** operations (by example):

```
myTable={'firstValue',2,3} -- builds a table with 3 values

print(myTable[1]) -- prints the first element in the table

table.insert(myTable,4) -- appends the number 4 to the table
```

- Concatenation (by example):

```
a='hello'
b=' world'
c=a..b -- c contains 'hello world'
```

- Length operator #:

```
stringLength=#'hello world'

tableSize=#{1,2,3,4,5}
```

Bitwise operators

- Lua supports the following bitwise operators:

- & bitwise AND
- | bitwise OR
- ~ bitwise exclusive OR
- >> right shift
- << unary bitwise NOT
- ~ unary bitwise NOT

Coroutines or threads

- Coroutines are easily created and resumed with:

```
-- Create a coroutine:
corout=coroutine.create(coroutineMain)

-- Start/resume a coroutine:
if coroutine.status(corout)~='dead' then
    local ok,errorMsg=coroutine.resume(corout)
    if errorMsg then
        error(debug.traceback(corout,errorMsg),2)
    end
end

-- The coroutine itself:
function coroutineMain()
    while not sim.getSimulationStopping() do
        -- some code
    end
end
```