



Die Skriptsprache Lua

Uwe Berger
bergeruw@gmx.net



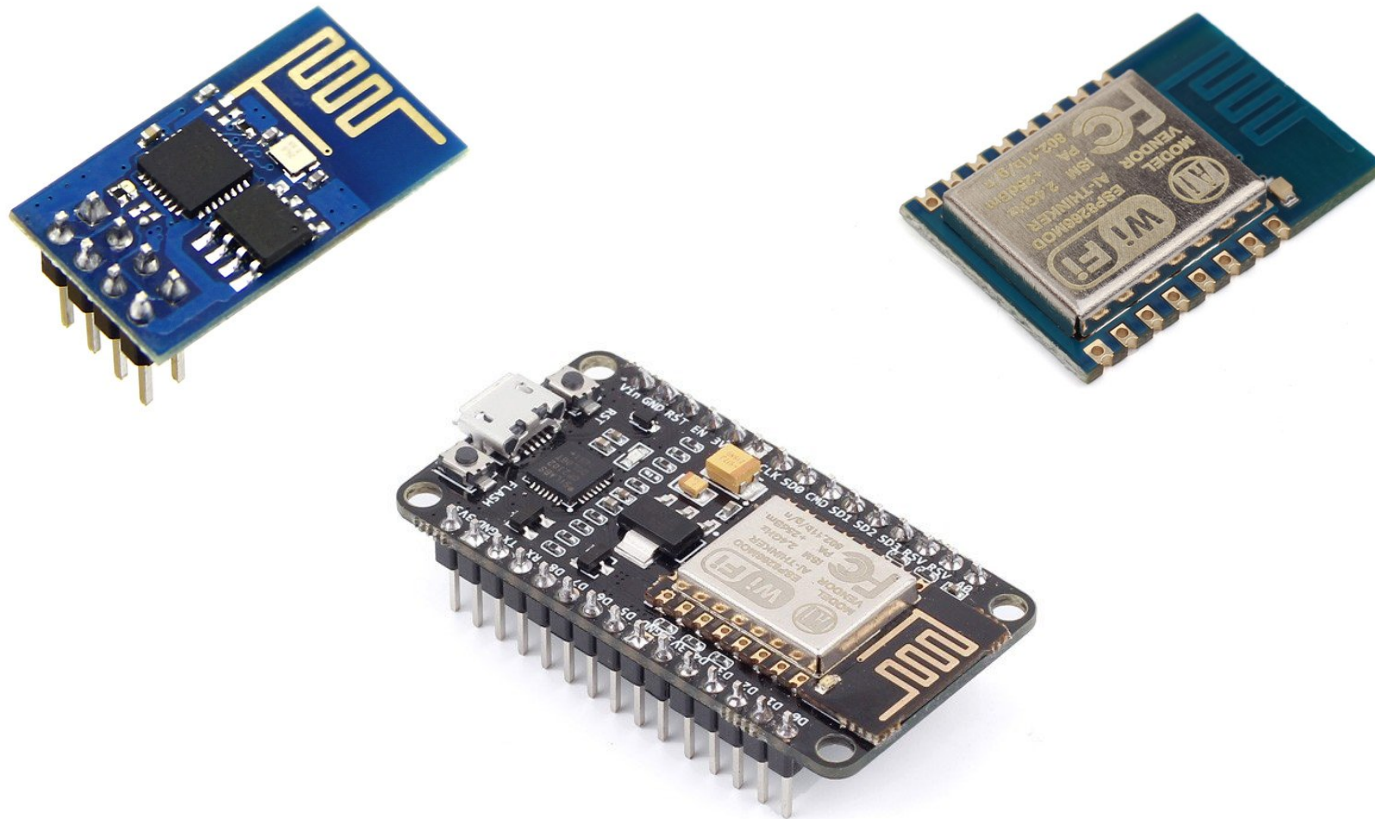
Uwe Berger



- Beruf: Softwareentwickler
- Freizeit: u.a. mit Hard- und Software rumspielen
- Linux seit ca. 1995
- BraLUG e.V.
- bergeruw@gmx.net



Meine Motivation...





Etwas Geschichte

- Erste Version: 1993
- Aktuelle Version: 5.3.3
- Computer Graphics Technology Group der Päpstlichen Katholischen Universität von Rio de Janeiro
- Roberto Ierusalimschy, Waldemar Celes, Luiz Henrique de Figueiredo
- Lizenz:
 - bis Version 4 → eigene BSD-Lizenz
 - ab Version 5 → MIT-Lizenz
- Webseite: <http://www.lua.org/>



Was ist Lua, wer benutzt es?

- Eine Skriptsprache mit folgenden Eigenschaften:
 - Erweiterbar
 - Einfach
 - Effizient
 - Portabel
- Lua-Benutzer benutzen...
 - Lua als C-Bibliothek
 - Lua als Erweiterungssprache in Anwendungen:
→ <http://www.lua.org/uses.html>
 - Lua als eigenständige Skriptsprache



Wer mitmachen möchte...

Browser:

- <http://www.lua.org/demo.html>
- <http://codepad.org/>

Kommandozeile:

```
$ lua
Lua 5.3.2 Copyright (C) 1994-2015 Lua.org, PUC-Rio
> print("Lua ist cool")
Lua ist cool!
^C
$

$ lua hello_world.lua
Hello World!
$
```



Einige Code-Beispiele...

- Lua-Manual:
→ <https://www.lua.org/manual/5.3/>
- Ausgewählte Skript-Beispiele:
→ <https://github.com/boerge42/Lua-Magie>



Einige lexikalische Konventionen

- Bezeichner: `[a-zA-Z_][a-zA-Z_0-9]*`
- Lua arbeitet „case-sensitive“
- Reservierte Schlüsselwörter sind:

```
and      elseif   function  local    repeat   until
break    end      goto      nil      return   while
do       false   if        not      then
else     for     in        or       true
```

- Kommentare:

```
-- Das ist ein einzeiliger Kommentar...

--[[ ...und dies hier ist
      ein mehrzeiliger
      Kommentar
--]]
```




Variablentypen

- Lua ist eine dynamisch typisierte Sprache
- Grundtypen sind *nil*, *boolean*, *number*, *string*, *function*, *table*, *thread*, *userdata*
- Ermittlung des (momentanen) Typs → *type()*

```
print(type(a))      -- nil

a = 42
print(type(a))     -- number

a = "Antwort " .. a
print(type(a))     -- string

a = (a==a)
print(type(a))     -- boolean

a=42; a=tostring(a)
print(type(a))     -- string
```



Operatoren

- Aritmetische Operatoren: +, -, *, /, ^, %, - (als Negation)
- Vergleichsoperatoren: <, >, <=, >=, ==, ~=
- Logische Operatoren: and, or, not
- Zusammensetzung: .. (Doppelpunkt)
- Längenoperator: #
- Vorrangregeln sind die Üblichen, wie bei anderen Sprachen... (Ausnahmen? → RTFM)



Zuweisung

```
a=1
b=a                -- b=1
c=a+b            -- c=2

s="Hallo"
t=s.." Welt!"    -- t="Hallo Welt!"
t=t.." "..a      -- t="Hallo Welt! 1"

-- mal etwas cooles...
a, b, c = 1, 2, 3 -- a=1, b=2, c=3
a, b, c = 1, 2   -- a=1, b=2, c=nil
a, b, c = 0      -- Achtung! a=0, b=nil, c=nil
a, b = 1, 2, 3   -- die 3 wird "vergessen"

a=1
b=2
a, b = b, a      -- a=2, b=1
```



Tabellen, Arrays, Listen

- Tabellen ist der Mechanismus zur Darstellung von Daten (Arrays, Mengen, Datensätze, -strukturen etc.)
- Tabellen werden durch assoziative Arrays dargestellt
- Assoziative Arrays werden durch Zahlen, Strings u.ä. indiziert
- Tabellen sind interne Lua-Objekte; Variablen verwalten nur Referenzen auf diese Objekte
- Existiert keine gültige Referenz mehr, gibt Lua automatisch den entsprechenden Speicher frei...
- ... → RTFM



Tabellen, Arrays, Listen

```
a={}  
  
z={5, 4, 3, 2, 1}  
zw={"eins", "zwei", "drei", "vier"}  
print(z[2], zw[#zw])           -- 4      vier  
  
a={1, 2, 3, 4}  
b=a  
a[3]=nil  
print(b[3])                   -- nil  
  
a={null=0}  
a["eins"]=1  
a.zwei=2  
print(a.null, a.eins, a["zwei"]) -- 0 1 2  
  
a={x=12, y=34}  
c={a, 42}  
print(c[1].x, c[2])          -- 12 42  
b={xy=a, z=56}  
print(b.xy.x, b["xy"].y, b.z) -- 12 34 56
```



Kommandozeile als Tabelle

```
> lua script.lua aaa bbb
```

```
-- script.lua
```

```
print(a[-1])  --> lua
```

```
print(a[0])   --> script.lua
```

```
print(a[1])   --> aaa
```

```
print(a[2])   --> bbb
```

```
---> uebergabeparameter.lua (https://github.com/boerge42/Lua-Magie/)
```



Kontrollstrukturen

```
-- if - then - (elseif) - else
--
if val == 42 then print("Das ist die Antwort!") end

if val ~= 42 then
    print(val.." ist ungleich 42")
else
    print(val.. "ist gleich 42")
end

if val == nil then
    print(val.." hat keinen Wert.")
elseif val == 42 then
    print(val.." ist die Antwort!")
elseif val == 0 then
    print("Der Wert ist Null.")
else
    print("War es alles nicht...!")
end

-- Achtung...
if val then print("val ist nicht nil|false!") end
```



Kontrollstrukturen

```
--  
-- while - do  
--  
a={"eins", "zwei", "drei", "vier"}  
i=1  
while a[i] do  
    print(a[i])  
    i=i+1  
end  
  
--  
-- repeat - until  
--  
x=42  
repeat  
    x=x/3  
    print(x)  
until x<1
```




Kontrollstrukturen

```
--  
-- numerisches for  
--  
for i=1, 10 do print(i) end  
  
for i=10, 1, -1 do print(i) end  
  
a={"eins", "zwei", "drei", "vier"}  
for i=1, #a, 1 do  
    print(a[i])  
end  
  
-- undefiniert, da Schleifenvariablen nur am Anfang gesetzt werden!  
for i=1, 10, 1 do  
    i=i+2  
end  
  
-- ebenfalls Achtung...!  
i=42  
for i=1, 10 do print(i) end  
print(i) -- 42, da i in for-Schleife lokal ist!
```



Kontrollstrukturen

```
--  
-- generisches for  
--  
z={eins=1, zwei=2, drei=3, vier=4, fuenf=5}  
for k, v in pairs(z) do  
    print(k, v)  
end  
  
wt={"So", "Mo", "Di", "Mi", "Do", "Fr", "Sa"}  
for i, v in ipairs(wt) do  
    print(i, v)  
end  
  
-- weitere Iterations-Funktionen in der Standardbibliothek:  
-- io.lines  
-- string.gmatch (siehe nächste Folie)  
--  
-- ...oder selbst schreiben, z.B. closure_generisches_for.lua
```



Iterations-Funktion *string.gmatch()*

```
t = "Das ist ein Text"
for s in string.gmatch(t, "%s+") do
    print(s)
End

-- Das
-- ist
-- ein
-- Text
```

- Pattern:
→ <https://www.lua.org/manual/5.3/manual.html#6.4.1>
- Möchte jemand ein wenig damit rumspielen?



Kontrollstrukturen

```
-- break
i=42
while true do
  i=i/2
  if i<1 then break end
  print(i)
end

-- return
function f(v)
  return v, v/2, v/3, v/4
end
print(f(42))

-- goto
i=42
do ::anfang::
  i=i/2
  print(i)
  if i>1 then goto anfang end
end
```



Kontrollstrukturen

- Es gibt kein switch – case in Lua
- Einfachste Alternative: `if...then...elseif...else...end`
- ...oder man spielt ein wenig rum → z.B. mit Tabellen



Übung: „99 bottles of beer...“

```
---> http://www.99-bottles-of-beer.net/
```

```
99 bottles of beer on the wall, 99 bottles of beer.  
Take one down and pass it around, 98 bottles of beer on the wall.
```

```
98 bottles of beer on the wall, 98 bottles of beer.  
Take one down and pass it around, 97 bottles of beer on the wall.
```

```
...
```

```
2 bottles of beer on the wall, 2 bottles of beer.  
Take one down and pass it around, 1 bottle of beer on the wall.
```

```
1 bottle of beer on the wall, 1 bottle of beer.  
Take one down and pass it around, no more bottles of beer on the wall.
```

```
No more bottles of beer on the wall, no more bottles of beer.  
Go to the store and buy some more, 99 bottles of beer on the wall.
```

```
---> hq9plus/99bottle.lua
```



Funktionen

```
-- klassisch...
function incr (x)
    return x+1
end

print(incr(42))           -- 43
print(incr(42, 21))      -- 43, zweites Argument wird irgnoriert

-- geht auch
incr=function(x) return x+1 end

print(incr(42))          -- 43

-- mehrere Rueckgabewerte
function foo (x)
    return x+1, x+2, x+3
end

print(foo(42))           -- 43 44 45
a, b, c = foo(42)
```



Funktionen

```
-- variadische Funktionen
function sum(...)
    local summe = 0
    for i, v in ipairs{...} do summe=summe+v end
    return summe
end

print(summe(1, 2, 3, 4, 5))           -- 15

-- benannte Argumente
function rename(args)
    return os.rename(args.old, args.new)
end

rename{new="neue.txt", old="alt.txt"}

-- Funktionsdefinitionen ueberschreiben
old_print=print
function print(v) old_print("neues print: "..v) end

print(42)                             -- neues print: 42
```




Global und Lokal

- Variablen und Funktionen können entweder global oder lokal gelten
- Global → ...
- Geltungsbereiche für lokale Variablen/Funktionen:
 - programm.lua, modul.lua
 - function() ... end
 - while ... do ...end, repeat ... until, if ... then ... else ... end
 - do ... end
- Lokale Deklarationen erleichtern Lua die interne, automatische, zyklische Speicherbereinigung



Global und Lokal

```
function f ()
    local a=42
    return a/3
end
a=1
print(f())           -- 14
print(a)            -- 1

a=42
if true then
    local a=1
    print(a)        -- 1
end
print(a)            -- 42

a=42
do
    local a=a+23
    print(a)        -- 65
end
print(a)            -- 42
```



Closure

- Beschreibt eine Funktion, die Zugriff auf ihren Erstellungskontext (Speicher, Zustand) enthält, der wiederum von extern nicht sichtbar ist, außer über eine weitere interne Funktion...

```
-- Closure
function counter()
    local i=0
    return function() i=i+1; return i end
end

c1=counter()
print(c1())          -- 1
print(c1())          -- 2

c2=counter()
print(c2())          -- 1
print(c1(), c2())    -- 3 2

---> closure_generisches_for.lua
```



Übung: HQ9+ - Interpreter mit Lua

- HQ9+: eine esoterische Programmiersprache
→ <https://de.wikipedia.org/wiki/HQ9%2B>
- Syntax:
 - H → „Hello World“
 - Q → Quine
 - 9 → „99 Bottles of beer...“
 - + → ein internes Register um 1 erhöhen
- Aufgabe:
 - Aufruf: `lua hq9plus.lua <hq9+ - code>`
 - Syntax überprüfen
 - Wenn Syntax i.O., dann HQ9+ - Code ausführen
 - → `hq9plus/hq9plus.lua`



Nebenbei: ein paar „echte“ Lua-Quines

```
---> https://de.wikipedia.org/wiki/Quine\_\(Computerprogramm\)
```

```
a="a=%c%s%c;io.write(string.format(a,34,a,34))";  
io.write(string.format(a,34,a,34))
```

```
---> https://github.com/MakeNowJust/quine/blob/master/quine.lua
```

```
s="s=%q;print(string.format(s,s));print(string.format(s,s))
```



load(), loadstring(), loadfile(), dofile()

- *load()*, *loadstring()*: einen String, welcher Lua-Code enthält, vorübersetzen, aber nicht ausführen
- *loadfile()*: dito, aber Lua-Code steht in einer Datei
- *dofile()*: wie *loadfile()*, aber Code wird auch ausgeführt

```
-- Dateiinhalt test.lua
function()
    print("Hallo FrosCon2017!")
end
```

```
-- eigentliches Programm...

-- ...test.lua laden
hallo=loadfile("test.lua")

hallo()      -- Hallo FrosCon2017!

i=0
f=load("i=i+1")
f(); print(i)    -- 1
f(); print(i)    -- 2
```



Nochmal loadstring()

```
-- Beispiel Lua-Tabelle von einem String einlesen...

s = "{v1=11, v2=42, v3=23}"

local f = assert(loadstring("return "..s))
tab = f()

=tab.v1      -- 11
=tab.v2      -- 42
=tab.v3      -- 23

---> https://github.com/boerge42/nodemcu\_scripts/tree/master/mqtt2oled
---> mqtt2oled.lua
---> Funktion: fill_lists()
```



Module

- Module → Bibliotheken
- können in Lua, aber auch als C-Bibliothek, vorliegen

```
-- Dateiinhalt modul.lua
local M={}

function M.incr(v, w)
  if w ~= nil then
    return v + w
  else
    return v + 1
  end
end

M.answer = 42

return M
```

```
-- eigentliches Programm...

-- ...modul.lua laden
my=require "modul"

print(my.answer)           -- 42

print(my.incr(42, 2))     -- 44
```




LuaRocks

- Paketverwaltungssystem für Lua-Module
- <http://luarocks.org>

```
$ sudo apt-get install luarocks
...

$ sudo luarocks install lua-cjson
...

$ luarocks list

Installed rocks:
-----
lua-cjson
  2.1.0-1 (installed) - /usr/local/lib/luarocks/rocks

$ lua
> require "cjson"
```



Performance

- Lua ist schnell...!

	100000 x 20!
C	~ 0,070s
Lua	~ 0,115s
Tcl	~ 7s

```
f = 20
c = 100000

-----

function fact(n)
    if n <= 1 then
        return n
    else
        return (n * fact(n-1))
    end
end

-----

print (fact(f))

ts_begin = os.clock()
for i = 1, c, 1 do
    fact(f)
end
ts_end = os.clock()

print(c.." x "..f.."!")
print((ts_end - ts_begin).."s")
```



Noch eine Übung: Kommando `wc` in Lua

- `wc` gibt (ohne weitere Optionen) für eine Textdatei aus:
 - Anzahl Zeilen
 - Anzahl Wörter
 - Anzahl Zeichen
- Tipp:

```
-- Datei einlesen
local f = assert(io.open(arg[1], "r"))
local text = f:read("*all")
f:close()
```



Was heute nicht angesprochen wurde...

- Die letzten „Feinheiten“ zu Strings, Tabellen etc.
- Metatabellen, Metamethoden
- Koroutinen und Threads
- Objektorientierte Programmierung
- Environments und `_G`
- Standardbibliotheken
- Debug-Interface
- C-API
- ...und einiges mehr → RTFM



Weiterführende Informationsquellen

- <http://www.lua.org/>
- „Programmieren in Lua“; Ierusalimschy, Roberto; Open Source Press, September 2006; ISBN 3937514228
- <https://github.com/boerge42/Lua-Magie>
- https://github.com/boerge42/nodemcu_scripts



Ende!