Lua - Erste Schritte in der Programmierung

Knut Lickert

7. März 2007

Dieser Text zeigt einige einfache Lua-Anweisungen und welchen Effekt sie haben. Weitere Informationen oder eine aktuelle Version dieses Dokumentes gibt es bei http://lua.lickert.net

Inhaltsverzeichnis

1 Zeichenketten und deren Ausgabe in Lua

- Bildschirmausgaben erfolgen mit dem Kommando *print*.
- Klammern um die auszugebenden Ausdruck sind zu empfehlen[1]
- Mehrere Parameter können duch Komma getrennt werden.
- Strings können mit ' und "definiert werden.

Beispielprogramm

```
1 print     ('1: Hello World')
2 print ("2: Hello World")
```

Das Ergebnis:
1: Hello World
2: Hello World

1.1 Zeilenumbrüche

Zeilenumbrüche werden mit $\backslash n$ oder \ und einem echten Zeilenumbruch eingefügt

Beispielprogramm

```
1 print  (  '1: Hello\
2 ________World'  )
3 print  (  "2: Hello\
4 ________World"  )
```

Das Ergebnis:
1: Hello
World
2: Hello
World

[1] Bei direkt folgendem String funktioniert es ohne, bei Variablen sind die Klammern zwingend

Beispielprogramm

```
print ( '5: Hello\n World' )
print ( "6: Hello\n World" )
```

Das Ergebnis:
5: Hello
World
6: Hello
World

In [[und]] geschachtelte Werte erlauben ebenfalls die Definition von Zeichenketten.

Beispielprogramm

```
print [[Hello World]]
```

Das Ergebnis:
Hello World

Beispielprogramm

```
print [[Hello World,
you are so nice today.]]
```

Das Ergebnis:
Hello World,
you are so nice today.

- Mehrzeilige Eingaben sind möglich.

Beispielprogramm

```
print [[
Hello World,
you are so nice today.]]
```

Das Ergebnis:
Hello World,
you are so nice today.

- Zeilenumbruch direkt nach [[wird geschluckt.

Beispielprogramm

```
print [[
Hallo Welt,
wie schön bist du [[doch]] heute]]
```

Das Ergebnis:
Hallo Welt,
wie schön bist du [[doch]] heute

- Verschachtelte Definitionen sind möglich, innere Zeichenketten werden übernommen.

1.2 Sonderzeichen

Jedes Zeichen kann mit \ und dem Oktalwert[2]des ASCII-Wertes des Zeichens definiert werden.

1.2.1 Benannte Sonderzeichen

Jedes der Zeichen kann mit \ und dem Kürzel generiert werden.

Zeichen	Funktion	Kürzel
\a	PC-Lautsprecher	BEL
\b	Rücktaste, Backspace	BS
\f	Form Feed	FF
\n	newline, neue Zeile	LF
\r	carriage return	CR
\t	(horizontal) Tabular	HT
\v	vertical tab	VT
\\	Backslash	
\’	single quote, einfaches Anführungszeichen	
\”	double quote, doppeltes Anführungszeichen	
\[	left square bracket, eckige Klammer links	
\]	right square bracket, eckige Klammer rechts	

Im folgenden Beispiel sind einige Teile auskommentiert, da diese keine sichtbare Ausgabe erzeugen (z.B. der PS-Pieps BEL).

Beispielprogramm

```
1  --~ print ( "“a", "“a" )
2  --~ print ( "“b", "“b" )
3  --~ print ( "“f", "“f" )
4  --~ print ( "“n", "“n" )
5  --~ print ( "“r", "“r" )
6  --~ print ( "“”", "“”" )
7  --~ print ( "“v", "“v" )
8  print ( "\"", "“" )
9  print ( "\'", "\'" )
10 print ( "\"", "“" )
11 print ( "\[", "\[" )
12 print ( "\]", "\]" )
```

[2]Bevor jemand lange suchen muss: Im ASCII Chart jeweils rechts unten (Quelle: http://dante.ctan.org/tex-archive/info/ascii.tex)

Das Ergebnis:
“” ”
“’ ’
“” ”
“[[
“]]

2 Operatoren

2.1 Grundrechenarten

Negative Zahlen werden mit einem vorgestellten - dargestellt.
Binäre Operatoren:

Operator	Beschreibung	Beispiel
+	Addition	a = b + c
-	Subtraktion (negatives Vorzeichen)	a = b - c
*	Multiplikation	a = b * c
/	Division	a= b / c
^	Potenzierung	a = b^c

Beispielprogramm

```
1 print( '2 + 3 =', 2 + 3 )
2 print( '2 - 3 =', 2 - 3 )
3 print( '2 * 3 =', 2 * 3 )
4 print( '2 / 3 =', 2 / 3 )
5 print( '2 ^ 3 =', 2 ^ 3 )
```

Das Ergebnis:
2 + 3 = 5
2 - 3 = -1
2 * 3 = 6
2 / 3 = 0.66666666666667
2 ^ 3 = 8

2.2 Vergleiche

= = Gleichheit

∼= Ungleichheit

< Kleiner

<= Kleiner gleich

> Größer

>= Größer gleich

Beispielprogramm

```
print( '2 == 3 ', 2 == 3 )
print( '2 ~= 3 ', 2 ~= 3 )
print( '2 > 3 ', 2 > 3 )
print( '2 < 3 ', 2 < 3 )
print( '2 >= 3 ', 2 >= 3 )
print( '2 <= 3 ', 2 <= 3 )
```

Das Ergebnis:

2 == 3 false
2 ~= 3 true
2 > 3 false
2 < 3 true
2 >= 3 false
2 <= 3 true

Beispielprogramm

```
print( "'A' == 'B' ", 'A' == 'B' )
print( "'A' ~= 'B' ", 'A' ~= 'B' )
print( "'A' > 'B' ", 'A' > 'B' )
print( "'A' < 'B' ", 'A' < 'B' )
print( "'A' >= 'B' ", 'A' >= 'B' )
print( "'A' <= 'B' ", 'A' <= 'B' )
```

Das Ergebnis:

'A' == 'B' false
'A' ~= 'B' true
'A' > 'B' false
'A' < 'B' true
'A' >= 'B' false
'A' <= 'B' true

Beispielprogramm

```
print( "'a' == 'A' ", 'a' == 'A' )
print( "'a' ~= 'A' ", 'a' ~= 'A' )
print( "'a' > 'A' ", 'a' > 'A' )
print( "'a' < 'A' ", 'a' < 'A' )
print( "'a' >= 'A' ", 'a' >= 'A' )
print( "'a' <= 'A' ", 'a' <= 'A' )
```

Das Ergebnis:
'a' == 'A' false
'a' ~= 'A' true
'a' ¿ 'A' true
'a' ¡ 'A' false
'a' ¿= 'A' true
'a' ¡= 'A' false

2.3 Logische Operatoren

not Logisches Nein

and Logisches und

or logisches or

Beispielprogramm

```
1 print( "true and false =", true and false )
2 print( "true or false =", true or false )
```

Das Ergebnis:
true and false = false
true or false = true

2.4 Geklammerte Logische Operationen

Beispielprogramm

```
1 print( " true or  false  and false  =", true or  false  and false
  )
2 print( " true or (false  and false) =", true or (false  and false) )
3 print( "(true or  false) and false  =", (true or  false) and false
  )
```

Das Ergebnis:
true or false and false = true
true or (false and false) = true
(true or false) and false = false

2.5 Logische Operatoren mit Verneinung (not)

Beispielprogramm

```
print( "not true and false ======" , not true and false )
print( "not (true and false) ====" , not (true and false) )
print( "true and not false ======" , true and not false )
print( "not true or false =======" , not true or false )
print( "not (true or false) =====" , not (true or false) )
print( "true or not false =======" , true or not false )
```

Das Ergebnis:
not true and false = false
not (true and false) = true
true and not false = true
not true or false = false
not (true or false) = false
true or not false = true

3 Variablen in Lua

- Variablen müssen nicht deklariert werden, sie werden bei Bedarf erzeugt.
- Variablen sind nicht Typgebunden, der Typ ist implizit vom Wert den sie vertreten definiert. Wird der Wert geändert, ändert sich der Typ der Variable.
- Variablen sind global, außer sie werden als lokal definiert. Näheres dazu bei *Funktionen*.
- Werte können folgende Typen annehmen:
  - Nil (Zugleich der Wert von nicht angelegter Variablen)
  - Zahlen
  - Literale (Zeichen, Buchstaben, Wörter, etc.)
  - Boolean (wahr/falsch bzw. true/false)
  - Tabellen
  - Funktionen (Spezialfall eines Blocks)

Beispielprogramm

```
print( var )
var = "Hello World"
print( var )
```

Das Ergebnis:
nil
Hello World

- Undefinierte Variablen ergeben *nil*
- Zuweisung erfolgt mit =

Beispielprogramm

```
print( var1, var2 )
var1, var2 = 1, 4
print( var1, var2 )
```

Das Ergebnis:
nil nil
1 4

- Mehrfachzuweisungen sind möglich.

4 Verzweigungen

4.1 If-Anweisung mit Else-Zweig

Beispielprogramm

```
a = 1
if a == 1 then
        print( 'a ist eins' )
else
        print( 'a ist nicht eins sondern ', a)
end
```

Das Ergebnis:
a ist eins

Beispielprogramm

```
a = 999
if a == 1 then
        print( 'a ist eins' )
else
        print( 'a ist nicht eins sondern ', a)
end
```

Das Ergebnis:
a ist nicht eins sondern 999

4.1.1 If mit verschiedenen Abfragen

Beispielprogramm

```
a = 2
if a == 1       then
        print( 'a ist eins' )
elseif a == 2 then
        print( 'a ist zwei' )
else
  print( 'a ist nicht eins oder zwei')
  print( 'a ist  ', a)
end
```

Das Ergebnis:
a ist zwei

4.2 Case-Anweisung

Eine Case-Anweisung existiert nicht, es muß durch if-elseif-Anweisungen simuliert werden.

5 Schleifen

5.1 For-Schleifen

Parameter der *For*-Anweisung:

1. Startwert
2. Endwert
3. Inkrement

Beispielprogramm

```
for variable = 0, 10, 2 do
        print ( variable )
end
```

Das Ergebnis:
0
2
4
6
8
10

Beispielprogramm

```
for variable = 0, 1, .5 do
        print ( variable )
end
```

Das Ergebnis:
0
0.5
1

Die Schleifenwerte müssen nicht ganzzahlig sein.

Beispielprogramm

```
for variable = 0, 1, .5 do
        print ( variable )
end
```

Das Ergebnis:
0
0.5
1

Ein Herunterzählen funktioniert ebenfalls.
Für Tabellen existiert eine Variante der *For*-Schleife.

5.2 While-Schleifen

Beispielprogramm

```
i = 1
while i  ¡= 5 do
         print (i)
         i = i + 1
end
```

Das Ergebnis:
1
2
3
4
5

- Anfangsbedingte Schleife
- Ist die Bedingung am Anfang nicht erfüllt, wird kein Block durchlaufen.

5.3 Repeat-Schleife

Beispielprogramm

```
1 i = 1
2 repeat
3         print (i)
4         i = i + 1
5 until i ¿ 5
```

Das Ergebnis:
1
2
3
4
5

- Endbedingte Schleife
- Mindestens ein Durchlauf des Blockes

5.4 Schleifenabbruch

Sollen Schleifen vorzeitig abgebrochen werden, kann *Break* verwendet werden.

Beispielprogramm

```
1 for variable = 1, -1, -.5 do
2         if variable == 0 then
3                 print "Null erreicht"
4                 break
5         end
6         print ( variable )
7 end
```

Das Ergebnis:
1
0.5
Null erreicht

6 Funktionen

Beispielprogramm

```
1 function Test()
2         print( "Hallo Welt" )
```

```
3 end
4
5 Test()
```

Das Ergebnis:
Hallo Welt

Beispielprogramm

```
1 function test()
2         return   "Hallo Welt"
3 end
4
5 print ( test() )
```

Das Ergebnis:
Hallo Welt

Rückgabewerte werden mit *return* angegeben.

Beispielprogramm

```
1 function test()
2         return   "Hallo", "Welt"
3 end
4
5 v1, v2 = test()
6 print( v1 )
7 print( v2 )
```

Das Ergebnis:
Hallo
Welt

Mehrfache Rückgabewerte sind möglich.

Beispielprogramm

```
1 function test()
2         return   "Hallo", "Welt"
3 end
4
5 v = test()
6 print( v )
```

Das Ergebnis:
Hallo

Werden mehrere Werte zurückgegeben, aber nicht entgegengenommen, dann werden die zusätzlichen Werte 'geschluckt'.

6.1 Funktionen mit Parameter

Beispielprogramm

```
function summe( v1, v2)
        return  ( v1 + v2 )
end

print( summe( 1, 2 ) )
print( summe( 2, 3 ) )
```

Das Ergebnis:
3
5

- Parameter beginnen laut Konvention mit _

Beispielprogramm

```
a = 'vorher'
function summe( v1, v2)
        a = v1 + v2
        return  a
end

print( summe( 1, 2 ) )
print( a )
print( v1 )
```

Das Ergebnis:
3
3
nil

- Parameter sind lokal definiert.
- Sonstige Variablen sind global definiert.

Beispielprogramm

```
a = 'vorher'
function summe( v1, v2)
```

```
3           local a = ˙v1 + ˙v2
4           return  a
5   end
6
7   print( summe( 1, 2 ) )
8   print( a )
9   print( ˙v1 )
```

Das Ergebnis:
3
vorher
nil

- Werden Variablen mit *local* definiert, sind sie nur innerhalb des Blockes verfügbar.

7 Tabellen und Listen

Lua-Tabellen sind Datensätze.[3]

Im Gegensatz zum gewohnten Tabellenbegriff sind sie keine Tabellen mit mehreren Zeilen und Spalten, sondern zweispaltige Tabellen mit Schlüssel und Wert.

Schlüssel	Wert
name	Gödel
Geburtsjahr	1906
Todesjahr	1978

7.1 Tabellen anlegen und füllen

Beispielprogramm

```
1   goedel = -"
2   goedel.name = "Kurt␣Gödel"
3   goedel.geburtsjahr = 1906
4   goedel.todesjahr         = 1978
5
6   print( goedel )
7   print( goedel.name )
```

Das Ergebnis:
table: 00329E38
Kurt Gödel

- Felder können an eine Tabelle angehängt werden.

[3] In Pascal ein Record, in Ruby ein Hash,...

- Tabellenname und Feldname sind durch einen Punkt getrennt.

Beispielprogramm

```
goedel = --
         name = "Kurt Gödel",
         geburtsjahr = 1906,
         todesjahr       = 1978,
"

print( goedel )
print( goedel.name )
```

Das Ergebnis:
table: 00329E38
Kurt Gödel

- Felder können beim Anlegen schon gefüllt werden.
- Komma nicht vergessen!

Beispielprogramm

```
goedel = --"
goedel.name = "Kurt Goedel"
goedel.geburtsjahr = 1906
goedel.todesjahr       = 1978
goedel[1] = 'Über die Vollständigkeit der Axiome des logischen Funktionenkalk
goedel[2] = 'Diskussion zur Grundlegung der Mathematik, Erkenntnis'

print( goedel )
print( goedel.name )
print( goedel[1] )
```

Das Ergebnis:
table: 00329E38
Kurt Goedel
Über die Vollständigkeit der Axiome des logischen Funktionenkalküls.

- Als Schlüsselwerte sind auch Zahlen möglich.
- Zahlen als Schlüssel sind in eckigen Klammern.
- Dieser Mechanismus ermöglicht die Definition von Listen.

Beispielprogramm

```
goedel = -"
goedel.name = "Kurt Gödel"

print( goedel )
print( goedel.name )
print( goedel["name"] )
```

Das Ergebnis:
table: 00329E38
Kurt Gödel
Kurt Gödel

- Statt der *.Feldname*-Methode kann auf Inhalte mit *[feldname]* erfolgen
- In den eckigen Klammern kann ein Ausdruck stehen (z.B. Variable mit dem Feldnamen)

7.2 Geschachtelte Tabellen

Tabellenwerte können wieder Tabellen sein.

Beispielprogramm

```
goedel = -"
goedel.name = "Kurt Gödel"
goedel.geburt   = -"
goedel.geburt.jahr              = 1906
goedel.geburt.monat     = 4
goedel.geburt.tag               = 28

goedel.gestorben                        = -"
goedel.gestorben.jahr           = 1978
goedel.gestorben.monat  = 1
goedel.gestorben.tag            = 14

print( goedel )
print( goedel.name )
print( goedel.geburt )
print( goedel.geburt.jahr )
```

Das Ergebnis:
table: 00329E38
Kurt Gödel
table: 00329FE8
1906

7.3 Tabellen durchlaufen

Die For-Schleife ermöglicht ein durchlaufen von Tabellen.

```
for schluessel, wert in pairs(table) do
        -- Block
end
```

- *schluessel* enthält den Feldnamen, bzw. bei Listen die Position.
- *wert* enthält den Wert.

Beispielprogramm

```
personen = {}
personen[1] = {
        name = "Kurt Gödel",
        geburtsjahr = 1906,
        todesjahr       = 1978,
}
personen[2] = {
        name = "Maurits Cornelis Escher",
        geburtsjahr = 1898 ,
        todesjahr       = 1972,
}

for variable, person in pairs(personen) do
        print( variable, person.name )
end
```

Das Ergebnis:
1 Kurt Gödel
2 Maurits Cornelis Escher

Eine Alternative Syntax ist:

```
for schluessel, wert in next, table do
        -- Block
end
```

Beispielprogramm

```
personen = {}
personen[1] = {
        name = "Kurt Gödel",
        geburtsjahr = 1906,
```

```
        todesjahr       = 1978,
"
personen[2] = -
        name = "Maurits Cornelis Escher",
        geburtsjahr = 1898 ,
        todesjahr       = 1972,
"

for variable, person in pairs(personen) do
        print( variable, person.name )
end
```

Das Ergebnis:
1 Kurt Gödel
2 Maurits Cornelis Escher

8 Beispiele mit regulären Ausdrücke

Beispielprogramm

```
s = "hello world from Lua"
for w in string.gfind(s, "%a+") do
        print(w)
end
```

Das Ergebnis:
hello
world
from
Lua

%a ist ein Pattern, das für Buchstaben steht. *+* wiederholt dieses. Ergebnis: Der String wird wortweise ausgegeben.

Beispielprogramm

```
function test.balanced.scan( s )
        print( '---->', s )
        for w in string.gfind(s, "%b{}") do
                print(w)
        end
end
test.balanced.scan( "Hallo {schöne und wunderbare} Welt" )
test.balanced.scan( "Hallo {schöne {und} wunderbare} Welt" )
```

Das Ergebnis:
----¿ Hallo –schöne und wunderbare″ Welt
–schöne und wunderbare″
----¿ Hallo –schöne –und″ wunderbare″ Welt
–schöne –und″ wunderbare″

Mit *%b* können geschachtelte Klammern analysiert werden. Üblicherweise würde im zweiten Aufruf

–schöne –und″

als Ergebnis kommen.

8.1 Patterns

Buchstaben definieren sich selbst, ausser sie sind eines der folgenden Zeichen:

ˆ$()%.[]*+-?

Es gibt folgende Zeichenklassen:

- . (ein Punkt) Ein beliebiges Zeichen.
- %a Ein beliebiger Buchstabe
- %c Ein Kontrollzeichen
- %d Eine beliebige Ziffer (digit)
- %l Ein beliebiger Kleinbuchstabe (lower case)
- %p Ein beliebiges Satzzeichen (punctuation)
- %s Ein Leerzeichen (space characters).
- %u Ein beliebiger Großbuchstabe (uppercase).
- %w Ein beliebiges alphanumerisches Zeichen (word-character)
- %x Eine beliebige Hexadezimalziffer.
- %z *represents the character with representation 0* ? nil?
- %x Das Zeichen *x* selbst. (Escape-Mechanismus für Pattern-Zeichen.