Lua-Dokumentation

1 Einführung

Lua ist eine Skriptsprache um allgemeine prozedurale Programmierung zu ermöglichen. Lua bietet darüber hinaus eine gute Unterstützung zur objektorientierten, funktionalen und datengestützten Programmierung. Lua ist dazu gedacht, als leistungsfähige und leichtgewichtige Skriptsprache für Programme genutzt zu werden, welche eine solche benötigen. Lua ist als Bibliothek in C implementiert.
Da Lua eine Skriptsprache ist, besitzt diese keine Art "Hauptprogramm": Sie arbeitet ausschließlich eingebettet in einer Host-Anwendung. Diese kann Funktionen aufrufen, um Lua-Code auszuführen, Lua-Variablen lesen und schreiben und C-Funktionen registrieren, welche per Lua-Code aufgerufen werden können. Durch das Verwenden von C-Funktionen kann Lua erweitert werden, um zahlreiche Anforderungen in verschiedenen Bereichen abzudecken und somit eine angepasste Programmiersprache mit einem gemeinsamen syntaktischen Framework zu verwenden. Die Lua-Distribution enthält ein Beispiel-Hostprogramm namens lua, welches mit Hilfe der Lua-Bibliothek einen eigenständigen Lua-Interpreter zur Verfügung stellt.
Lua ist freie Software und wird wie üblich und in der Lizenz beschrieben ohne Gewährleistung angeboten. Die Implementierung, welche in dieser Dokumentation beschrieben wird, ist auf der offiziellen Webpräsenz von Lua (www.lua.org) verfügbar. Für eine Diskussion zu den Entscheidungen des Lua-Designs sehen Sie sich die technischen Dokumente, welche auf der Webpräsenz von Lua verfügbar sind, an. Für eine detaillierte Einführung in die Programmierung mit Lua sehen Sie sich Robertos Buch "Programming in Lua" (zweite Auflage) an.

2 Die Sprache

Dieser Abschnitt beschreibt den Wortschatz, die Syntax und die Semantik von Lua. Mit anderen Worten: Dieser Abschnitt beschreibt, welche Zeichen gültig sind, wie diese kombiniert werden können und was diese Kombinationen bedeuten.
Die Sprachkonstrukte werden durch Verwendung der üblichen EBNF erklärt, in welcher {a} 0 oder mehrere a und [a] ein optionales a bedeutet. Nicht-Terminale werden als Nicht-Terminale dargestellt, Schlüsselworte als Schlüsselwort und andere Terminale wie `=´. Die komplette Syntax von Lua findet sich unter §?8am Ende dieser Dokumentation.

2.1 Lexikalische Konventionen

Bezeichner (auch Identifikatoren genannt) können in Lua eine beliebige Zeichenkette aus Buchstaben, Ziffern und Unterstrichen sein, dürfen jedoch nicht mit einer Ziffer beginnen. Dies entspricht der Definition eines Bezeichners der meisten Sprachen. (Die Definition eines Buchstabens hängt von der aktuellen Sprache ab: Jedes Zeichen, welches von der aktuellen Sprache als alphabetisch angesehen wird, kann in einem Bezeichner verwendet werden.) Identifikatoren werden benutzt, um Variablen und Tabellenfelder zu benennen.
Die folgenden Schlüsselworte sind reserviert und können nicht als Bezeichner verwendet werden:
and break do else elseif
end false for function if
in local nil not or
repeat return then true until while
Lua ist eine Sprache, welche zwischen Groß-/Kleinschreibung unterscheidet: and ist ein reserviertes Wort, aber And und AND sind zwei verschiedene, gültige Bezeichner. Per Konvention sind Bezeichner, welche mit einem Unterstrich beginnen und im Folgenden durchgehend großgeschrieben werden (so wie _VERSION), reserviert und werden für interne globale Variablen von Lua verwendet.
Die folgenden Zeichenketten stehen für weitere Token:
+ - * / % ^ #
== ~= <= >= < > =
( ) { } [ ]
; : , ...

Literale können durch übereinstimmende einfache oder doppelte Anführungszeichen begrenzt werden und die folgenden C-ähnlichen Steuerzeichen enthalten: '\a' (bell), '\b' (backspace), '\f' (form feed), '\n' (newline), '\r' (carriage return), '\t' (horizontal tab), '\v' (vertical tab), '\\' (Backslash), '\"' (Anführungszeichen [doppelt]) und '\'' (Apostroph [einfach]). Darüber hinaus erzeugt ein Backslash gefolgt von einem echten Zeilenumbruch einen Zeilenumbruch in der Zeichenkette. Ein Zeichen in einer Zeichenkette kann ebenso durch dessen numerischen Wert über die Maskierung \ddd angegeben werden, wobei ddd eine Sequenz von bis zu drei Dezimalziffern darstellt. (Beachten Sie, dass wenn eine numerische Maskierung von einer Ziffer gefolgt wird, diese durch die Verwendung von exakt drei Ziffern ausgedrückt werden muss.) Zeichenketten in Lua können beliebige 8-bit-Werte enthalten – inklusive Nullen, welche per '\0' notiert werden können.
Literale können durch die Benutzung von langen Klammern auch in einem längeren Format definiert werden. Wir definieren eine öffnende lange Klammer der Ebene n als eine öffnende eckige Klammer, gefolgt von n Gleichheitszeichen, gefolgt von einer weiteren öffnenden eckigen Klammer. Eine öffnende lange Klammer der Ebene 0 wird somit als [[geschrieben, eine öffnende lange Klammer der Ebene 1 als [=[usw. Eine schließende lange Klammer ist analog dazu definiert; eine schließende lange Klammer der Ebene 4 wird beispielsweise als]====] geschrieben. Eine lange Zeichenkette beginnt mit einer öffnenden langen Klammer beliebiger Ebene und endet bei der ersten schließenden langen Klammer der gleichen Ebene. Literale in dieser geklammerten Form können sich über mehrere Zeilen erstrecken, interpretieren keine Steuerzeichen und ignorieren lange Klammern anderer Ebene. Sie können Beliebiges enthalten, außer eine schließende Klammer der entsprechenden Ebene.
Der Bequemlichkeit wegen wird ein Zeilenumbruch, der unmittelbar auf eine öffnende lange Klammer folgt, nicht in die Zeichenkette mit aufgenommen. Beispielsweise entsprechen in einem ASCII-System (bei welchem 'a' als 97 kodiert wird, ein Zeilenumbruch als 10 und '1' als 49) die folgenden fünf Literale alle der gleichen Zeichenkette:

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

Eine numerische Konstante kann mit einem optionalen Dezimalteil und einem optionalen dezimalen Exponenten geschrieben werden. Lua akzeptiert auch ganzzahlige hexadezimale Konstanten mit dem Präfix 0x. Beispiele gültiger numerischer Konstanten sind ...

```
3 3.0 3.1416 314.16e-2 0.31416E1 0xff 0x56
```

Ein Kommentar beginnt mit zwei Bindestrichen (--) irgendwo außerhalb einer Zeichenkette. Sofern der Text direkt nach -- keine öffnende lange Klammer ist, handelt es sich um einen kurzen Kommentar, welcher bis zum Ende der Zeile geht. Andernfalls handelt es sich um einen langen Kommentar, der sich bis zur entsprechenden schließenden langen Klammer erstreckt. Lange Kommentare werden häufig benutzt, um Code temporär zu deaktivieren.

2.2 Werte und Typen

Lua ist eine dynamisch typisierte Sprache. Das bedeutet, dass Variablen keine Typen besitzen, sondern nur Werte einen solchen haben. Es gibt keine Typdefinitionen in der Sprache. Alle Werte besitzen ihren eigenen Typ.
Alle Werte in Lua sind Werte erster Ordnung. Das bedeutet, dass alle Werte in Variablen gespeichert, als Argumente an Funktionen übergeben, oder als Ergebnis zurückgegeben werden können.
Es gibt acht Basistypen in Lua: nil, boolean, number, string, function, userdata, thread und table. Nil ist der Typ des Werts nil, dessen Haupteigenschaft es ist, verschieden von allen anderen Werten zu sein; für gewöhnlich zeigt er die Abwesenheit eines sinnvollen

Wertes an. Boolean ist der Typ der Werte false und true. Sowohl nil als auch false machen eine Bedingung falsch; jeder andere Wert macht sie wahr. Number repräsentiert reelle (doppelt genaue Gleitpunkt-)Zahlen. (Es ist einfach, Lua-Interpreter zu erstellen, welche andere interne Repräsentationen für Zahlen wie z. B. Gleitkommazahlen mit einfacher Genauigkeit oder lange Ganzzahlen nutzen; s. Datei luaconf.h.) String repräsentiert ein Feld von Zeichen. Lua benutzt 8 Bit: Zeichenketten können beliebige 8-bit-Zeichen enthalten, inklusive Nullen ('\0') (s. §?2.1).
Lua kann in Lua oder C geschriebene Funktionen aufrufen und manipulieren (s. §?2.5.8). Der Typ userdata wird angeboten, um beliebige C-Daten in Lua-Variablen zu speichern. Dieser Typ entspricht einem Block rohen Speichers und hat außer der Zuweisung und einem Gleichheitstest keine vordefinierten Operationen in Lua. Durch die Verwendung von Metatabellen kann der Programmierer jedoch Operationen für "userdata"-Werte definieren (s. §?2.8)."Userdata"-Werte können in Lua nicht erstellt oder bearbeitet werden, sondern nur über die C-API. Dies garantiert die Integrität der Daten des Host-Programms.
Der Typ thread repräsentiert unabhängige Threads zur Ausführung und wird zur Implementierung von Koroutinen benutzt (s. §?2.11).Verwechseln Sie nicht Lua-Threads mit denen des Betriebssystems. Lua unterstützt unter allen Systemen Koroutinen; auch auf Systemen, die keine Threads unterstützen.
Der Typ table implementiert assoziative Felder; dies sind Felder, welche nicht nur mit Zahlen, sondern mit beliebigen Werten (außer nil) indiziert werden können. Tabellen können heterogen sein, d. h. diese können Werte aller Typen (außer nil) enthalten. Tabellen sind der grundlegende Mechanismus in Lua für Datenstrukturen; diese können benutzt werden, um gewöhnliche Felder, Symboltabellen, Mengen, Strukturen, Graphen, Bäume etc. zu repräsentieren. Um Strukturen darzustellen benutzt Lua den Feldbezeichner als Index. Die Sprache unterstützt diese Repräsentation durch a.name als eine syntaktische Vereinfachung für a["name"]. Es gibt einige bequeme Wege, um Tabellen in Lua zu erzeugen (s. §?2.5.7).
Genauso wie Indizes können die Werte eines Tabellenfeldes von beliebigem Typ (außer nil) sein. Im Besonderen können Tabellenfelder Funktionen enthalten, da diese Werte erster Ordnung sind. Derartige Tabellen können ebenso Methoden beinhalten (s. §?2.5.9).

Tabellen, Funktionen, Threads und Benutzerdaten-Werte sind Objekte: Die Variablen enthalten diese Werte eigentlich nicht, sondern lediglich Referenzen auf diese. Zuweisung, Parameterübergabe und Funktionen liefern immer eine Referenz auf diese Werte; diese Operationen implizieren keinerlei Form des Kopierens.
Die Bibliotheksfunktion type liefert eine Zeichenkette zur Beschreibung des Typs eines übergebenen Wertes.

2.2.1 Typumwandlung

Lua bietet eine automatische Konvertierung von Zeichenketten und Zahlenwerten zur Laufzeit. Jede arithmetische Operation, welche auf eine Zeichenkette angewendet wird, versucht diese Zeichenkette nach den üblichen Umwandlungsregeln zu konvertieren. Umgekehrt wird immer, wenn eine Zahl benutzt wird, wo eine Zeichenkette erwartet wird, diese Zahl zu einer Zeichenkette in einem vernünftigen Format konvertiert. Für eine komplette Kontrolle darüber, wie Zahlen zu Zeichenketten konvertiert werden, benutzen Sie die format-Funktion aus der Zeichenkettenbibliothek (s. string.format).

2.3 Variablen

Variablen sind Orte, an denen Werte gespeichert werden. Es gibt drei Arten von Variablen in Lua: globale Variablen, lokale Variablen und Tabellenfelder.
Ein einzelner Bezeichner kann eine globale Variable oder eine lokale Variable bezeichnen (oder den Parameter einer Funktion, welcher eine besondere Form einer lokalen Variablen darstellt):
var ::= Name
Name entspricht einem Bezeichner wie unter §?2.1definiert.
Jede Variable wird als global angenommen, solange diese nicht explizit als lokale Variable deklariert wird (s. §?2.4.7).Lokale Variablen sind lexikalisch sichtbar: Auf lokale Variablen kann von Funktionen, die in deren Sichtbarkeitsbereich definiert sind, frei

zugegriffen werden. (s. §?2.6).
Vor einer ersten Zuweisung an eine Variable ist deren Wert nil.
Eckige Klammern werden benutzt, um eine Tabelle zu indizieren:
var ::= prefixexp `[´ exp `]´
Die Bedeutung des Zugriffs auf globale Variablen und Tabellenfelder kann über Meta-Tabellen geändert werden. Der Zugriff auf eine indizierte Variable t[i] ist äquivalent zum Aufruf gettable_event(t,i). (s. §?2.8für eine komplette Beschreibung der gettable_event-Funktion. Diese Funktion ist nicht in Lua definiert oder aufrufbar. Wir benutzen diese hier lediglich zu Erklärungszwecken.)
Die Syntax var.Name ist lediglich eine syntaktische Vereinfachung von var["Name"]:
var ::= prefixexp `.´ Name
Alle globalen Variablen befinden sich als Felder in gewöhnlichen Lua-Tabellen, genannt Umgebungstabellen oder einfach Umgebungen. (s. §?2.9).Jede Funktion hat ihre eigene Referenz zu einer Umgebung, so dass alle globalen Variablen in dieser Funktion auf diese Umgebungstabelle zeigen werden. Wenn eine Funktion erzeugt wird, so erbt diese die Umgebung der Funktion, welche sie erzeugt hat. Um die Umgebungstabelle einer Lua-Funktion zu erhalten, rufen Sie getfenv auf. Um sie zu ersetzen, rufen Sie setfenv auf. (Sie können die Umgebung von C-Funktionen nur über die Debug-Bibliothek manipulieren (s. §?5.9).)
Ein Zugriff auf eine globale Variable x ist äquivalent zu _env.x, welches wiederum äquivalent ist zu …
gettable_event(_env,"x")
… wobei _env die Umgebung der laufenden Funktion ist. (s. §?2.8für eine komplette Beschreibung der gettable_event-Funktion. Diese Funktion ist nicht in Lua definiert oder aufrufbar. Genauso ist die _env-Variable nicht in Lua definiert. Wir benutzen diese hier lediglich zu Erklärungszwecken.)

2.4 Anweisungen

Lua unterstützt eine weitestgehend konventionelle Menge an Anweisungen, ähnlich denen von Pascal oder C. Dies beinhaltet Zuweisungen, Kontrollstrukturen, Funktionsaufrufe und Variablendeklarationen.

2.4.1 Chunks

Eine Ausführungseinheit in Lua wird Chunk genannt. Ein Chunk ist ganz einfach eine Folge von Anweisungen, welche sequentiell ausgeführt werden. Jede Anweisung kann optional von einem Semikolon gefolgt werden:
chunk ::= {stat [`;´]}
Es gibt keine leeren Anweisungen und somit ist ';;' nicht legal.
Lua behandelt einen Chunk als Inhalt einer anonymen Funktion mit einer variablen Anzahl an Argumenten (s. §?2.5.9).Somit können Chunks lokale Variablen definieren, Argumente erhalten und Werte zurückgeben.
Ein Chunk kann in einer Datei oder Zeichenkette des Host-Programms gespeichert werden. Um einen Chunk auszuführen, kompiliert Lua diesen zu Anweisungen für eine virtuelle Maschine vor und führt dann den kompilierten Code mit einem Interpreter für diese virtuelle Maschine aus.
Chunks können auch in eine binäre Form vorkompiliert werden; s. Programm luac für Details. Programme als Quelltext oder in kompilierter Form sind austauschbar; Lua erkennt den Typ automatisch und handelt entsprechend.

2.4.2 Blöcke

Ein Block ist eine Liste von Anweisungen; syntaktisch ist ein Block dasselbe wie ein Chunk:
block ::= chunk
Ein Block kann explizit begrenzt werden, um eine einzelne Anweisung zu erzeugen:
stat ::= do block end
Explizite Blöcke sind nützlich, um den Gültigkeitsbereich von Variablen zu steuern. Explizite Blöcke werden auch manchmal benutzt, um eine return- oder break-Anweisung innerhalb eines anderen Blocks hinzuzufügen (s. §?2.4.4).

2.4.3 Zuweisungen

Lua erlaubt mehrfache Zuweisungen. Somit definiert die Syntax von für Zuweisungen eine Liste von Variablen auf der linken Seite und eine Liste von Ausdrücke auf der rechten Seite. Die Elemente beider Listen werden per Komma getrennt:
stat ::= varlist `=´ explist
varlist ::= var {`,´ var}
explist ::= exp {`,´ exp}
Ausdrücke werden unter §?2.5erläutert.
Vor der Zuweisung wird die Liste der Werte auf die Länge der Liste der Variablen angepasst. Falls es mehr Werte als benötigt gibt, werden die überschüssigen Werte verworfen. Falls es weniger Werte als benötigt gibt, wird die Liste um eine entsprechende Anzahl nil erweitert. Falls die Liste der Ausdrücke mit einem Funktionsaufruf endet, gehen alle daraus resultierenden Rückgabewerte vor der Anpassung in die Liste der Werte ein (außer, wenn der Aufruf geklammert ist; s. §?2.5).
Die Zuweisung wertet zuerst alle ihre Ausdrücke aus und erst dann werden die Zuweisungen durchgeführt. Im Code …
i = 3
i,a[i] = i+1,20
… wird a[3] ohne a[4] zu beeinflussen auf 20 gesetzt, weil das i in a[i] zuerst (zu 3) ausgewertet wird, bevor es 4 zugewiesen bekommt. Ähnlich werden in der Zeile …
x,y = y,x
… die Werte von x und y ausgetauscht und …
x,y,z = y,z,x
… führt eine zyklische Permutation der Werte von x, y und z durch.
Die Bedeutung von Zuweisungen an globale Variablen oder Tabellenfelder kann über Metatabellen geändert werden. Eine Zuweisung an eine indizierte Variable t[i] = val ist äquivalent zu settable_event(t,i,val). (S. §?2.8für eine komplette Beschreibung der settable_event-Funktion. Diese Funktion ist in Lua nicht definiert oder aufrufbar. Wir benutzen sie hier lediglich zu Erklärungszwecken.)
Eine Zuweisung an eine globale Variable x = val ist äquivalent zu der Zuweisung _env.x = val, welche wiederum äquivalent zu …
settable_event(_env,"x",val)
… ist, wobei _env die Umgebung der laufenden Funktion darstellt. (Die _env-Variable ist in Lua nicht definiert. Wir benutzen sie hier lediglich zu Erklärungszwecken.)

2.4.4 Kontrollstrukturen

Die Kontrollstrukturen if, while und repeat haben die gewöhnliche Bedeutung und bekannte Syntax:
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block} [else block] end
Lua besitzt auch eine for-Anweisung in zwei Varianten (s. §?2.4.5).
Die Bedingung einer Kontrollstruktur kann beliebige Werte zurückgeben. Sowohl false als auch nil werden als falsch angenommen. Alle anderen – von nil und false verschiedenen – Werte werden als wahr angenommen (insbesondere sind die Nummer 0 und die leere Zeichenkette ebenfalls wahr).
Bei der repeat–until-Schleife endet der innere Block nicht beim until-Schlüsselwort, sondern nach der Bedingung. Somit kann die Bedingung auf lokale Variablen verweisen, welche innerhalb des Schleifenblocks deklariert sind.
Die return-Anweisung wird benutzt, um Werte aus einer Funktion oder einem Chunk (welcher ebenfalls nur eine Funktion ist) zurückzugeben. Funktionen und Chunks können mehr als einen Wert zurückgeben und somit ist die Syntax für die return-Anweisung …
stat ::= return [explist]
Die break-Anweisung wird benutzt, um die Ausführung einer while-, repeat- oder for-Schleife zu beenden und zur nächsten Anweisung nach der Schleife zu springen:
stat ::= break
Ein break beendet die innerste verschachtelte Schleife.
Die return- und break-Anweisungen können nur als letzte Anweisung eines Blocks geschrieben werden. Falls es wirklich notwendig ist, return oder break innerhalb eines Blocks zu verwenden, kann ein expliziter innerer Block wie bei den Idiomen do return end

und do break end verwendet werden, weil nun return und break die letzten Anweisungen in deren (inneren) Blöcken sind.

2.4.5 FOR-Anweisung

Die for-Anweisung hat zwei Formen: Eine numerische und eine generische.
Die numerische for-Schleife wiederholt einen Code-Block, während eine Laufvariable eine arithmetische Folge durchläuft. Sie hat folgende Syntax:
stat ::= for Name `=´ exp `,´ exp [`,´ exp] do block end
block wird für Name beginnend beim Wert des ersten exp wiederholt, bis es mit der Schrittweite des dritten exp das zweite exp erreicht. Genauer gesagt ist eine for-Anweisung wie …

```
for v = e1, e2, e3 do block end
```

… äquivalent zu …

```
do
local var, limit, step = tonumber(e1), tonumber(e2), tonumber(e3)
if not (var and limit and step) then error() end
while (step 0 and var = limit) do
local v = var
block
var = var + step
end
end
```

Beachten Sie Folgendes:

Alle drei Ausdrücke zur Steuerung werden lediglich einmal ausgewertet, bevor die Schleife beginnt. Sie müssen alle in einer Zahl resultieren.
var, limit und step sind unsichtbare Variablen. Die Bezeichner dienen hier lediglich Erklärungszwecken.
Falls kein dritter Ausdruck (die Schrittweite) angegeben wird, wird eine Schrittweite von 1 benutzt.
Sie können break benutzen, um eine for-Schleife zu verlassen.
Die Schleifenvariable v ist lokal für die Schleife; Sie können deren Wert nach Beendigung oder Abbruch von for nicht verwenden. Falls Sie diesen Wert benötigen, weisen Sie ihn vor dem Abbruch oder der Beendigung der Schleife einer anderen Variable zu.

Die generische for-Anweisung arbeitet mit Funktionen, genannt Iteratoren. Nach jeder Iteration wird der Iterator aufgerufen, um einen neuen Wert zu erzeugen, bis der neue Wert nil ist. Die generische for-Schleife hat folgende Syntax:
stat ::= for namelist in explist do block end
namelist ::= Name {`,´ Name}
Eine for-Anweisung wie …

```
for var_1, ···, var_n in explist do block end
```

… ist äquivalent zu …

```
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
```

Beachten Sie Folgendes:

2.4.6 Funktionsaufrufe als Anweisungen

Um Seiteneffekte zu ermöglichen, können Funktionsaufrufe als Anweisung ausgeführt werden:

stat ::= functioncall
In diesem Fall werden alle zurückgegebenen Werte verworfen. Funktionsaufrufe werden unter §?2.5.8beschrieben.

2.4.7 Lokale Deklarationen

Lokale Variablen können überall innerhalb eines Blocks deklariert werden. Die Deklaration kann eine initiale Zuweisung beinhalten:
stat ::= local namelist [`=´ explist]
Sofern gegeben hat eine initiale Zuweisung die gleiche Semantik wie eine mehrfache Zuweisung (s. §?2.4.3).Andernfalls werden alle Variablen mit nil initialisiert.
Ein Chunk ist auch ein Block (s. §?2.4.1),wodurch lokale Variablen auch innerhalb eines Chunks, außerhalb eines expliziten Blocks, deklariert werden können.
Die Sichtbarkeitsregeln für lokale Variablen sind unter §?2.6beschrieben.

2.5 Ausdrücke

Die Basis-Ausdrücke in Lua lauten wie folgt:
exp ::= prefixexp
exp ::= nil | false | true
exp ::= Number
exp ::= String
exp ::= function
exp ::= tableconstructor
exp ::= `...´
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | `(´ exp `)´
Zahlen und Literale werden unter §?2.1erklärt; Variablen werden unter §?2.3erklärt; Funktionsdefinitionen werden unter §?2.5.9erklärt; Funktionsaufrufe werden unter §?2.5.8erklärt; Tabellenkonstruktoren werden unter §?2.5.7erklärt. Ausdrücke mit variabler Anzahl an Argumenten – geschrieben durch drei Punkte ('...') – können nur direkt innerhalb einer solchen Funktion verwendet werden; diese werden unter §?2.5.9 erklärt.
Binäre Operatoren beinhalten arithmetische Operatoren (s. §?2.5.1),relationale Operatoren (s. §?2.5.2),logische Operatoren (s. §2.5.3) und den Konkatenierungsoperator (s. §?2.5.4).Unäre Operatoren enthalten das unäre Minus (s. §?2.5.1),das unäre not (s. §?2.5.3)und den unären Längenoperator (s. §?2.5.5).
Sowohl Funktionsaufrufe als auch Ausdrücke mit variabler Anzahl an Argumenten können in mehreren Werten resultieren. Wenn ein Ausdruck als Anweisung benutzt wird (nur für Funktionsaufrufe möglich, s. §?2.4.6)),wird deren Rückgabeliste auf 0 Elemente begrenzt und verwirft somit alle Rückgaben. Falls ein Ausdruck als letztes (oder einziges) Element einer Liste von Ausdrücken benutzt wird, findet keine Anpassung statt (sofern der Aufruf nicht geklammert ist). In allen anderen Fällen wird das Ergebnis auf ein Element begrenzt, wobei alle Werte außer dem ersten verworfen werden.
Hier sind ein paar Beispiele:

```
f() -- liefert keine Ergebnisse
g(f(), x) -- f() liefert ein Ergebnis
g(x, f()) -- g erhält x und alle Ergebnisse von f()
a,b,c = f(), x -- f() liefert ein Ergebnis (c wird nil)
a,b = ... -- a erhält den ersten variablen Parameter, b erhält den
-- zweiten (sowohl a als auch b können nil werden, wenn
-- kein entsprechender variabler Parameter existiert)
a,b,c = x, f() -- f() liefert zwei Ergebnisse
a,b,c = f() -- f() liefert drei Ergebnisse
return f() -- liefert alle Ergebnisse von f()
return ... -- liefert alle erhaltenen variablen Parameter
return x,y,f() -- liefert x, y und alle Ergebnisse von f()
{f()} -- erzeugt eine Liste aus allen Ergebnissen von f()
{...} -- erzeugt eine Liste aus allen variablen Parametern
{f(), nil} -- f() liefert ein Ergebnis
```

Jeder geklammerte Ausdruck liefert jeweils genau einen Wert. Insofern ist (f(x,y,z)) immer ein einzelner Wert, auch wenn f mehrere Werte liefert. (Der Wert von (f(x,y,z)) ist der erste von f gelieferte Wert oder nil, wenn f keinerlei Werte zurückgibt.)

2.5.1 Arithmetische Operatoren

Lua unterstützt die geläufigen arithmetischen Operatoren: das binäre + (Addition), - (Subtraktion), * (Multiplikation), / (Division), % (Modulo) und ^ (Exponentiation) sowie das unäre - (Negation). Sofern die Operanden Zahlen – oder Zeichenketten, welche zu Zahlen konvertiert werden können (s. §?2.2.1)– sind, so haben alle Operationen die geläufige Bedeutung. Exponentiation funktioniert mit allen Exponenten; beispielsweise berechnet x^(-0.5) das Inverse der Quadratwurzel aus x. Modulo ist wie folgt definiert:
a % b == a - math.floor(a/b)*b
Dies ist der Rest der Division, welche den Quotienten auf -Unendlich rundet.

2.5.2 Relationale Operatoren

Die relationalen Operatoren in Lua sind …
== ~= =
Diese Operatoren resultieren immer in false oder true.
Gleichheit (==) vergleicht zuerst die Typen der Operanden. Falls die Typen unterschiedlich sind, ist das Ergebnis false. Andernfalls werden die Werte der Operanden verglichen. Zahlen und Zeichenketten werden auf gewöhnliche Weise verglichen. Objekte (Tabellen, Benutzerdaten, Threads und Funktionen) werden per Referenz verglichen: Zwei Objekte werden nur als gleich angenommen, wenn sie das selbe Objekt sind. Jedes mal, wenn Sie ein neues Objekt (eine Tabelle, Benutzerdaten, einen Thread oder eine Funktion) erzeugen, ist dieses Objekt von bisher existierenden Objekten verschieden. Sie können die Art, wie Lua Tabellen und Benutzerdaten vergleicht durch die Verwendung der "eq"-Metamethode (s. §?2.8)ändern.
Die Konvertierungsregeln aus §?2.2.1beziehen sich nicht auf Gleichheitsprüfungen. Insofern evaluiert "0"==0 zu false und t[0] und t["0"] bezeichnen verschiedene Einträge einer Tabelle.
Der Operator ~= ist exakt die Negation der Gleichheit (==).
Die Richtungsoperatoren arbeiten wie folgt: Wenn beide Argumente Zahlen sind, werden sie als solche verglichen. Andernfalls, wenn beide Argumente Zeichenketten sind, werden deren Werte entsprechend der aktuellen Sprache verglichen. Andernfalls versucht Lua, die "lt"- oder "le"-Metamethoden (s. §?2.8)aufzurufen. Ein Vergleich der Art a b wird zu b übersetzt und a = b zu b .

2.5.3 Logische Operatoren

Die logischen Operatoren in Lua sind and, or und not. Wie die Kontrollstrukturen (s. §?2.4.4)betrachten alle logischen Operatoren false sowie nil als false und alles andere als true.
Der Negationsoperator not gibt immer false oder true zurück. Der Konjunktionsoperator and gibt sein erstes Argument zurück, wenn dieser false oder nil ist; andernfalls gibt and sein zweites Argument zurück. Der Disjunktionsoperator or gibt sein erstes Argument zurück, wenn dieser von nil und false verschieden ist; andernfalls gibt or sein zweites Argument zurück. Beide verwenden eine Kurzschlussauswertung, was bedeutet, dass das zweite Argument nur falls nötig ausgewertet wird. Hier ein paar Beispiele:

```
10 or 20 -- 10
10 or error() -- 10
nil or "a" -- "a"
nil and 10 -- nil
false and error() -- false
false and nil -- false
false or nil -- nil
10 and 20 -- 20
```

(In dieser Referenz zeigt -- das Ergebnis des vorhergehenden Ausdrucks an.)

2.5.4 Konkatenierung

Der Operator zur Konkatenierung von Zeichenketten in Lua wird mit zwei Punkten ("..") notiert. Wenn beide Operanden Zeichenketten oder Zahlen sind, so werden diese zu

Zeichenketten entsprechend der unter §?2.2.1erwähnten Vorgehensweise konvertiert. Andernfalls wird die "concat"-Metamethode aufgerufen (s. §?2.8).

2.5.5 Der Längenoperator

Der Längenoperator wird als unärer Operator # geschrieben. Die Länge einer Zeichenkette entspricht ihrer Anzahl an Bytes (dies ist die gewöhnliche Bedeutung der Länge einer Zeichenkette, wenn jedes Zeichen ein Byte belegt).
Die Länge einer Tabelle t ist als ganzzahliger Index n definiert, so dass t[n] nicht nil ist und t[n+1] nil ist; darüber hinaus kann n 0 sein, wenn t[1] nil ist. Bei einem regulären Feld mit nicht-nil Werten von 1 bis zu einem gegebenen n ist dessen Länge eben dieses n – der Index des letzten Wertes. Falls das Feld "Löcher" hat (d. h. nil-Werte zwischen anderen nicht-nil Werten), kann #t irgendeiner der Indizes sein, welcher direkt einem nil-Wert vorangeht (d. h. er nimmt einen solchen nil-Wert als Ende des Feldes an).

2.5.6 Präzedenz

Die Operatorpräzedenz in Lua entspricht der unten angegebenen Tabelle; von niedriger zu hoher Priorität:
or
and
= ~= ==
..
+ -
* / %
not # - (unär)
^
Wie üblich können Sie Klammern zur Änderung der Auswertungsreihenfolge eines Ausdrucks verwenden. Die Konkatenierung ("..") und Exponentiation ("^") sind rechtsassoziativ. Alle anderen binären Operatoren sind linksassoziativ.

2.5.7 Tabellen-Konstruktoren

Tabellen-Konstruktoren sind Ausdrücke, welche eine Tabelle erzeugen. Jedes mal, wenn ein Konstruktor ausgewertet wird, wird eine neue Tabelle erzeugt. Ein Konstruktor kann benutzt werden, um eine leere Tabelle zu erzeugen, oder um eine Tabelle zu erzeugen und einige ihrer Felder zu initialisieren. Die allgemeine Syntax von Konstruktoren lautet:
tableconstructor ::= `{´ [fieldlist] `}´
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= `[´ exp `]´ `=´ exp | Name `=´ exp | exp
fieldsep ::= `,´ | `;´
Jedes Feld der Form [exp1] = exp2 fügt einer neuen Tabelle einen Eintrag mit dem Schlüssel exp1 und dem Wert exp2 hinzu. Ein Feld der Form name = exp ist äquivalent zu ["name"] = exp. Schließlich sind Felder der Form exp äquivalent zu [i] = exp, wobei i fortlaufende Ganzzahlen beginnend bei 1 sind. Felder in anderen Formaten beeinflussen diese Zählung nicht. Beispiel:

```
a = { [f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45 }
```

... ist äquivalent zu ...

```
do
local t = {}
t[f(1)] = g
t[1] = "x" -- erster Ausdruck
t[2] = "y" -- zweiter Ausdruck
t.x = 1 -- t["x"] = 1
t[3] = f(x) -- dritter Ausdruck
t[30] = 23
t[4] = 45 -- vierter Ausdruck
a = t
end
```

Falls das letzte Feld der Liste die Form exp hat und der Ausdruck ein Funktionsaufruf oder ein Ausdruck mit variabler Anzahl Argumente ist, werden alle von diesem Ausdruck zurückgegebenen Werte fortlaufend der Liste hinzugefügt (s. §?2.5.8).Um dies zu verhindern, können Sie den Funktionsaufruf oder Ausdruck klammern (s. §?2.5).

Die Feldliste kann – zur Bequemlichkeit bei automatisch erzeugtem Code – optional einen abschließenden Separator enthalten.

2.5.8 Funktionsaufrufe

Ein Funkionsaufruf in Lua hat folgende Syntax:
functioncall ::= prefixexp args
Bei einem Funktionsaufruf werden zuerst "prefixexp" und "args" ausgewertet. Falls der Wert von "prefixexp" vom Typ function ist, wird diese Funktion mit den übergebenen Argumenten aufgerufen. Andernfalls wird die call-Metamethode mit dem Wert von "prefixexp" als erstem Parameter gefolgt von den originalen Argumenten aufgerufen (s. §?2.8).
Die Form …
functioncall ::= prefixexp `:´ Name args
… kann benutzt werden, um "Methoden" aufzurufen. Ein Aufruf der Art v:name(args) ist eine syntaktische Vereinfachung für v.name(v,args), mit der Ausnahme, dass v nur einmal ausgewertet wird.
Argumente haben folgende Syntax:
args ::= `(´ [explist] `)´
args ::= tableconstructor
args ::= String
Alle Argument-Ausdrücke werden vor dem Aufruf ausgewertet. Ein Aufruf der Art f{fields} ist eine syntaktische Vereinfachung für f({fields}); dies bedeutet, dass die Argumentliste eine einzelne neue Tabelle ist. Ein Aufruf der Art f'string' (oder f"string" oder f[[string]]) ist eine syntaktische Vereinfachung für f('string'); dies bedeutet, dass die Argumentliste ein einzelner Literal ist.
Als Ausnahme zur frei formatierbaren Syntax von Lua gilt, dass Sie keinen Zeilenumbruch vor einer '(' in einem Funktionsaufruf verwenden können. Diese Einschränkung verhindert einige Doppeldeutigkeiten in der Sprache. Falls Sie …

```
a = f
(g).x(a)
```

… schreiben, würde Lua dies als einzelne Anweisung a = f(g).x(a) interpretieren. Falls Sie also zwei Anweisungen möchten, müssen Sie ein Semikolon zwischen diese hinzufügen. Wenn Sie eigentlich f aufrufen möchten, müssen Sie den Zeilenumbruch vor (g) entfernen.
Ein Aufruf der Art return functioncall wird Endrekursion genannt. Lua implementiert endständige Funktionen mit konstantem Speicherplatzverbrauch: In einer Endrekursion verwendet die aufgerufene Funktion den Stapelspeicher der aufrufenden Funktion. Deshalb gibt es keine Begrenzung der Anzahl verschachtelter Endrekursionen, welche ein Programm ausführen kann. Eine Endrekursion leert jedoch jegliche Debug-Informationen der aufrufenden Funktion. Beachten Sie, dass eine Endrekursion nur bei einer bestimmten Schreibweise geschieht, bei welcher return einen einzelnen Funktionsaufruf als Argument besitzt; diese Syntax veranlasst die aufrufende Funktion genau die Rückgabe der aufgerufenen Funktion zu liefern. Folgende Beispiele sind demnach keine Endrekursionen:

2.5.9 Funktionsdefinitionen

Die Syntax zur Funktionsdefinition lautet wie folgt:
function ::= function funcbody
funcbody ::= `(´ [parlist] `)´ block end
Die folgenden syntaktischen Vereinfachungen machen dem Umgang mit Funktionsdefinitionen leichter:
stat ::= function funcname funcbody
stat ::= local function Name funcbody
funcname ::= Name {`.´ Name} [`:´ Name]
Die Anweisung …
function f () body end
… wird übersetzt zu …
f = function () body end
Die Anweisung …

function t.a.b.c.f () body end
... wird übersetzt zu ...
t.a.b.c.f = function () body end
Die Anweisung ...
local function f () body end
... wird übersetzt zu ...
local f; f = function () body end
... und nicht zu ...
local f = function () body end
(Das macht lediglich einen Unterschied, wenn der Inhalt der Funktion Referenzen auf f enthält.)
Ein Funktionsdefinition ist ein ausführbarer Ausdruck, dessen Wert vom Typ function ist. Sobald Lua einen Chunk vorkompiliert, werden all dessen Funktionsrümpfe ebenfalls vorkompiliert. Sobald Lua die Funktion ausführt, wird diese instanziiert. Diese Funktionsinstanz (oder Closure) ist der finale Wert des Ausdrucks. Verschiedene Instanzen der selben Funktion können auf verschiedene externe lokale Variablen verweisen und verschiedene Umgebungstabellen haben.
Parameter agieren als lokale Variablen, welche mit den Argumentwerten initialisiert werden:
parlist ::= namelist [`,´ `...´] | `...´
Wenn eine Funktion aufgerufen wird, wird deren Liste von Argumenten auf die Länge der Liste der Parameter angepasst, sofern es sich nicht um eine Funktion mit variabler Argumentanzahl handelt, welche durch drei Punkte ('...') am Ende der Parameterliste angezeigt wird. Eine Funktion mit variabler Argumentanzahl passt ihre Argumentliste nicht an; stattdessen sammelt diese alle ihre Argumente und stellt sie der Funktion über einen Ausdruck variabler Argumentanzahl, welcher ebenso mit drei Punkten geschrieben wird, zur Verfügung. Der Wert dieses Ausdrucks ist eine Liste aller zusätzlichen Argumente, ähnlich einer Funktion mit mehreren Rückgabewerten. Wenn ein Ausdruck mit variabler Argumentanzahl in einem anderen Ausdruck oder innerhalb einer Liste von Ausdrücken genutzt wird, wird dessen Rückgabeliste auf ein Element angepasst. Wenn der Ausdruck als letztes Element einer Liste von Ausdrücken benutzt wird, wird keine Anpassung durchgeführt (sofern der letzte Ausdruck nicht geklammert ist).
Betrachten Sie beispielsweise folgende Definition:
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
Dies ergibt folgendes Mapping der Argumente zu Parametern und zu dem Ausdruck mit variabler Argumentanzahl:
CALL PARAMETERS
f(3) a=3, b=nil
f(3, 4) a=3, b=4
f(3, 4, 5) a=3, b=4
f(r(), 10) a=1, b=10
f(r()) a=1, b=2
g(3) a=3, b=nil, ... -- (nothing)
g(3, 4) a=3, b=4, ... -- (nothing)
g(3, 4, 5, 8) a=3, b=4, ... -- 5 8
g(5, r()) a=5, b=1, ... -- 2 3
Ergebnisse werden durch die Benutzung der return-Anweisung zurückgegeben (s. §?2.4.4).Falls das Ende der Funktion erreicht wird, ohne eine return-Anweisung vorzufinden, liefert die Funktion keine Ergebnisse zurück.
Die Punktnotation wird zur Definition von Methoden benutzt, d. h. für Funktionen, welche einen impliziten zusätzlichen Parameter self besitzen. Insofern ist die Anweisung ...
function t.a.b.c:f (params) body end
... eine syntaktische Vereinfachung für ...

2.6 Sichtbarkeitsregeln

Lua ist eine Sprache mit lexikalischer Sichtbarkeit. Der Gültigkeitsbereich einer Variablen beginnt mit der ersten Anweisung nach deren Deklaration und bleibt bis zum Ende des

innersten Blocks der in der Deklaration enthalten ist. Gegeben sei folgendes Beispiel:

```
x = 10 -- globale Variable
do -- neuer Block
local x = x -- neues 'x' mit dem Wert 10
print(x) -- 10
x = x+1
do -- ein weiterer Block
local x = x+1 -- ein weiteres 'x'
print(x) -- 12
end
print(x) -- 11
end
print(x) -- 10 (die globale)
```

Beachten Sie, dass sich bei einer Deklaration wie local x = x das neu deklarierte x noch nicht in einem Gültigkeitsbereich befindet und somit das zweite x auf die äußere Variable zeigt.
Auf Grund der lexikalischen Sichtbarkeitsregeln kann auf lokale Variablen von Funktionen, welche innerhalb deren Gültigkeitsbereich definiert sind, frei zugegriffen werden. Eine lokale Variable, welche von einer inneren Funktion benutzt wird, wird innerhalb dieser freie Variable oder externe lokale Variable genannt.
Beachten Sie, dass jede Ausführung einer local-Anweisung neue lokale Variablen definiert. Gegeben sei folgendes Beispiel:

```
a = {}
local x = 20
for i=1,10 do
local y = 0
a[i] = function () y=y+1; return x+y end
end
```

Die Schleife erzeugt zehn Closures (d. h. zehn Instanzen der anonymen Funktion). Jede von diesen benutzt eine andere y-Variable, während alle das gleiche x teilen.

2.7 Fehlerbehandlung

Auf Grund der Tatsache, dass Lua eine eingebettete Skriptsprache ist, werden alle Lua-Aktionen durch C-Code des Hostprogramms durch einen Aufruf einer Funktion der Lua-Bibliothek ausgelöst (s. lua_pcall). Immer, wenn ein Fehler während der Kompilierung oder Ausführung von Lua auftritt, wird die Kontrolle an C zurückgegeben, welches entsprechend reagieren kann (evtl. eine Fehlermeldung ausgeben).
Lua-Code kann explizit eine Fehlermeldung durch einen Aufruf der error-Funktion erzeugen. Falls Sie Fehler in Lua abfangen möchten, können Sie die pcall-Funktion verwenden.

2.8 Metatabellen

Jeder Wert in Lua kann eine Metatabelle besitzen. Diese Metatabelle ist eine gewöhnliche Lua-Tabelle, welche das Verhalten des originalen Wertes bei bestimmten Operationen bestimmt. Sie können diverse Aspekte des Verhaltens von Operationen auf Werte durch das Setzen spezifischer Felder in dessen Metatabelle verändern. Wenn beispielsweise ein nicht-numerischer Wert der Operand einer Addition ist, sucht Lua nach einer Funktion im Feld "__add" in dessen Metatabelle. Wenn es eine findet, ruft Lua diese Funktion auf, um die Addition durchzuführen.
Wir nennen die Schlüssel einer Metatabelle Ereignisse und die Werte Metamethoden. Im vorherigen Beispiel ist das Ereignis "add" und die Metamethode ist die Funktion, welche die Addition durchführt.
Sie können die Metatabelle jedes Wertes über die getmetatable-Funktion abfragen.
Sie können die Metatabellen von Tabellen über die setmetatable-Funktion ersetzen. Sie können die Metatabellen anderer Typen von Lua nicht ändern (außer über die Debugbibliothek); Sie müssen hierfür die C-API verwenden.
Tabellen und alle Benutzerdaten haben individuelle Metatabellen (obwohl mehrere Tabellen und Benutzerdaten deren Metatabellen teilen können). Werte aller anderen Typen teilen sich eine einzige Metatabelle pro Typ; d. h. es gibt eine einzige Metatabelle

für alle Zahlen, eine für alle Zeichenketten etc.
Eine Metatabelle steuert, wie sich ein Objekt bei arithmetischen Operationen, Sortierungsvergleichen, Konkatenierungen, Längenoperationen und der Indizierung verhält. Eine Metatabelle kann darüber hinaus eine Funktion definieren, welche aufgerufen wird, wenn Benutzerdaten automatisch bereinigt werden. Für jede dieser Operationen assoziiert Lua einen spezifischen Schlüssel, welcher Ereignis genannt wird. Wenn Lua eine dieser Operationen auf einen Wert anwendet, prüft es, ob dieser Wert eine Metatabelle mit dem entsprechenden Ereignis besitzt. Falls dies der Fall ist, steuert der mit diesem Schlüssel verknüpfte Wert (die Metamethode) , wie Lua diese Operation durchführt.
Metatabellen steuern die im Folgenden aufgelisteten Operationen. Jede Operation wird durch ihren entsprechenden Bezeichner identifiziert. Der Schlüssel für jede Operation ist eine Zeichenkette mit zwei vorangestellten Unterstrichen ('__'); der Schlüssel für die Operation "add" ist beispielsweise die Zeichenkette "__add". Die Semantik dieser Operationen werden durch eine Lua-Funktion, welche beschreibt, wie der Interpreter Operationen ausführt, besser erklärt.
Der hier gezeigte Lua-Code dient lediglich der Illustration; das echte Verhalten ist fest im Interpreter kodiert und ist wesentlich effizienter als diese Simulation. Alle Funktionen, welche in diesen beschreibungen verwendet werden (rawget, tonumber etc.) sind unter §?5.1beschrieben. Insbesondere um die Metamethode eines gegebenen Objekts zu erhalten, benutzen wir den Ausdruck …
metatable(obj)[event]
Dies sollte wie folgt gelesen werden:
rawget(getmetatable(obj) or {},event)
Das bedeutet, dass der Aufruf einer Metamethode keine anderen Metamethoden aktiviert und der Zugriff auf Objekte ohne Metatabellen nicht fehlschlägt (sondern einfach in nil resultiert).

"add": Die +-Operation.
Die unten angegebene Funktion getbinhandler definiert, wie Lua den Handler für eine binäre Operation auswählt. Zuerst versucht Lua den ersten Operanden. Wenn dessen Typ keinen Handler für die Operation definiert, versucht Lua den zweiten Operanden.

```
function getbinhandler(op1,op2,event)
return metatable(op1)[event] or metatable(op2)[event]
end
```

Durch die Verwendung dieser Funktion ist das Verhalten von op1 + op2 …

```
function add_event(op1,op2)
local o1,o2 = tonumber(op1),tonumber(op2)
if o1 and o2 then -- sind beide Operanden numerisch?
return o1 + o2 -- '+' ist das primitive 'add'
else -- zumindest einer der Operanden ist nicht numerisch
local h = getbinhandler(op1,op2,"__add")
if h then
-- rufe den Handler mit beiden Operanden auf
return (h(op1,op2))
else -- kein Handler verfügbar: Standardverhalten
error(···)
end
end
end
```

"sub": Die --Operation. Verhalten analog zur "add"-Operation.

"mul": Die *-Operation. Verhalten analog zur "add"-Operation.

"div": Die /-Operation. Verhalten analog zur "add"-Operation.

"mod": Die %-Operation. Verhalten analog zur "add"-Operation, mit o1 - floor(o1/o2)*o2 als primitive Operation.

"pow": Die ^-Operation. Verhalten analog zur "add"-Operation, mit pow (aus der mathematischen Bibliothek von C) als primitive Operation.

"unm": Die unäre --Operation.

```
function unm_event(op)
local o = tonumber(op)
if o then -- Operand numerisch?
return -o -- '-' ist das primitive 'unm'
else -- der Operand ist nicht numerisch
-- versuche, einen Handler vom Operanden zu erhalten
local h = metatable(op).__unm
if h then
-- rufe den Handler mit dem Operanden auf
return (h(op))
else -- kein Handler verfügbar: Standardverhalten
error(···)
end
end
end
```

"concat": Die ..-Operation (Konkatenierung).

```
function concat_event(op1,op2)
if (type(op1) == "string" or type(op1) == "number") and
(type(op2) == "string" or type(op2) == "number") then
return op1 .. op2 -- primitive Zeichenketten-Verknüpfung
else
local h = getbinhandler(op1,op2,"__concat")
if h then
return (h(op1, op2))
else
error(···)
end
end
end
```

"len": Die #-Operation:

```
function len_event(op)
if type(op) == "string" then
return strlen(op) -- primitive Zeichenketten-Länge
elseif type(op) == "table" then
return #op -- primitive Tabellen-Länge
else
local h = metatable(op).__len
if h then
-- Handler mit Operand aufrufen
return (h(op))
else -- kein Handler verfügbar: Standardverhalten
```

```
error(···)
end
end
end
```
S. §?2.5.5für eine Beschreibung der Länge einer Tabelle.

"eq": Die ==-Operation. Die Funktion getcomphandler definiert, wie Lua eine Metamethode für Vergleichsoperationen auswählt. Eine Metamethode wird nur ausgewhlt, wenn beide Objekte vom selben Typ sind und die selbe Metamethode für die gewählte Operation haben.

```
function getcomphandler(op1,op2,event)
if type(op1) ~= type(op2) then return nil end
local mm1 = metatable(op1)[event]
local mm2 = metatable(op2)[event]
if mm1 == mm2 then return mm1 else return nil end
end
```

Das "eq"-Ereignis ist wie folgt definiert:

```
function eq_event(op1,op2)
if type(op1) ~= type(op2) then -- verschiedene Typen?
return false -- verschiedene Objekte
end
if op1 == op2 then -- primitiv gleich?
return true -- Objekte sind gleich
end
-- Metamethode versuchen
local h = getcomphandler(op1,op2,"__eq")
if h then
return (h(op1, op2))
else
return false
end
end
```

a ~= b ist äquivalent zu not (a == b).

"lt": Die <-Operation.

```
function lt_event(op1,op2)
if type(op1) == "number" and type(op2) == "number" then
return op1 < op2 -- numerischer Vergleich
elseif type(op1) == "string" and type(op2) == "string" then
return op1 < op2 -- lexikographischer Vergleich
else
local h = getcomphandler(op1,op2,"__lt")
if h then
return (h(op1,op2))
else
error(···)
end
end
end
```

a > b ist äquivalent zu b < a.

"le": Die <=-Operation.

```
function le_event(op1,op2)
if type(op1) == "number" and type(op2) == "number" then
return op1 <= op2 -- numerischer Vergleich
```

```
elseif type(op1) == "string" and type(op2) == "string" then
return op1 <= op2 -- lexikographischer Vergleich
else
local h = getcomphandler(op1,op2,"__le")
if h then
return (h(op1,op2))
else
h = getcomphandler(op1,op2,"__lt")
if h then
return not h(op2,op1)
else
error(···)
end
end
end
end
```

a >= b ist äquivalent zu b <= a. Beachten Sie, dass Lua in Abwesenheit einer "le"-Metamethode "lt" versucht – unter der Annahme, dass a <= b äquivalent zu not (b < a) ist.

2.9 Umgebungen

Neben Metatabellen haben Objekte des Typs thread, function oder userdata eine weitere Tabelle, welche mit diesen assoziiert ist und deren Umgebung genannt wird. Ähnlich Metatabellen, sind Umgebungen reguläre Tabellen und mehrere Objekte können die gleiche Umgebung teilen.

Threads werden erstellt, indem die Umgebung des erstellenden Threads geteilt wird. Benutzerdaten und C-Funktionen werden erstellt, indem die Umgebung der erstellenden C-Funktion geteilt wird. Nicht-verschachtelte Lua-Funktionen (durch loadfile, loadstring oder load erzeugt) werden durch Teilen der Umgebung des erstellenden Threads erstellt. Verschachtelte Lua-Funktionen werden durch Teilen der Umgebung der erstellenden Lua-Funktion erstellt.

Umgebungen, welche mit Benutzerdaten verknüpft werden, haben für Lua keine Bedeutung. Dies ist lediglich eine bequeme Funktionalität für Programmierer, um eine Tabelle an Benutzerdaten zu knüpfen.

Umgebungen, welche mit Threads verknüpft werden, werden globale Umgebungen genannt. Diese werden als standardmäßige Umgebung für Threads und nicht-verschachtelte Lua-Funktionen verwendet, welche vom Thread erzeugt werden und können direkt durch C-Code angesteuert werden (s. §?3.3).

Die Umgebung, welche mit einer C-Funktion verknüpft ist, kann direkt durch C-Code angesteuert werden (s. §?3.3).Diese wird als standardmäßige Umgebung für andere C-Funktionen und von der Funktion erzeugte Benutzerdaten verwendet.

Umgebungen, welche mit einer Lua-Funktion verknüpft sind, werden benutzt, um alle Zugriffe auf globale Variablen aus dieser Funktion heraus aufzulösen (s. §?2.3).Diese werden als standardmäßige Umgebung für verschachtelte Lua-Funktion, welche von der Funktion erstellt werden, genutzt.

Sie können die Umgebung einer Lua-Funktion oder eines laufenden Threads durch einen Aufruf von setfenv ändern. Sie können die Umgebung einer Lua-Funktion oder eines laufenden Threads durch einen Aufruf von getfenv erhalten. Um die Umgebung anderer Objekte (Benutzerdaten, C-Funktionen oder andere Threads) zu verändern, müssen Sie die C-API verwenden.

2.10 Automatische Speicherbereinigung

Lua führt eine automatische Speicherverwaltung durch. Dies bedeutet, dass Sie sich weder um die Anforderung von Speicher für neue Objekte, noch um dessen Freigabe, wenn die Objekte nicht länger benötigt werden, kümmern müssen. Lua verwaltet den Speicher automatisch durch eine automatische Speicherbereinigung, welche von Zeit zu Zeit alle "toten" Objekte (d. h. Objekte, welche nicht länger für Lua verfügbar sind) sammelt. Jeder Speicher, welcher von Lua verwendet wird, ist der automatischen Verwaltung unterworfen: Tabellen, Benutzerdaten, Funktionen, Threads, Zeichenketten

etc.
Lua implementiert einen inkrementellen Mark-and-Sweep-Algorithmus. Dieser verwendet zwei Nummern, um die Durchführung der automatische Speicherbereinigung zu steuern: Den "garbage-collector pause" und den "garbage-collector step multiplier". Beide verwenden Prozentpunkte als Einheit (so dass ein Wert von 100 einem internen Wert von 1 entspricht).
"garbage-collector pause" steuert, wie lange die automatische Speicherbereinigung bis zum nächsten Durchlauf wartet. Größere Werte machen die Bereinigung weniger agressiv. Werte kleiner als 100 bedeuten, dass die Bereinigung nicht auf den Beginn eines neuen Durchlaufs warten wird. Ein Wert von 200 bedeutet, dass die Bereinigung wartet, bis sich der insgesamt genutzte Speicher verdoppelt hat, bevor ein neuer Durchlauf startet .
"step multiplier" steuert die relative Geschwindigkeit der Bereinigung im Verhältnis zur Speicheranforderung. Größere Werte machen die Bereinigung agressiver, erhöhen aber auch die Größe jeder inkrementellen Schrittweite. Werte kleiner als 100 machen die Bereninigung langsam und können darin enden, dass die Bereinigung einen Vorgang niemals beendet. Der Standardwert 200 bedeutet, dass die Bereinigung "doppelt" so schnell wie die Speicheranforderung erfolgt.
Sie können diese Nummern durch einen Aufruf von lua_gc in C oder collectgarbage in Lua ändern. Mit diesen Funktionen können Sie die Bereinigung auch direkt steuern (z. B. anhalten und neu starten).

2.10.1 Metamethoden

Durch Verwendung der C-API können Sie Metamethoden zur automatischen Speicherbereinigung von Benutzerdaten setzen (s. §?2.8).Diese Metamethoden werden auch Finalisierer genannt. Finalisierung ermöglicht es, Lua's Mechanismus zur automatischen Speicherbereinigung in Bezug auf externe Resourcen zu koordinieren (so etwas wie Dateien, Netzwerk- oder Datenbankverbindungen schließen oder den eigenen Speicher leeren).
Verwaiste Benutzerdaten mit einem Feld __gc in deren Metatabellen werden nicht sofort vom Garbage-Collector gesammelt. Stattdessen fügt Lua diese in eine Liste ein. Nach der Bereinigung führt Lua eine äquivalente Funktion wie die folgende für alle Benutzerdaten in der Liste durch:

```
function gc_event (udata)
local h = metatable(udata).__gc
if h then
h(udata)
end
end
```

Am Ende jedes Durchlaufs der automatischen Speicherbereinigung werden die Finalisierer für Benutzerdaten in der umgekehrten Reihenfolge ihrer Erstellung aufgerufen, inklusive denen, welche in diesem Durchlauf gesammelt wurden. D. h., dass der erste Finalisierer der aufgerufen wird dieser ist, welcher den letzten erzeugten Benutzerdaten des Programms zugewiesen wurde. Die Benutzerdaten selbst werden erst im nächsten Durchlauf der automatischen Speicherbereinigung durchgeführt.

2.10.2 Weak-Tabellen

Eine Weak-Tabelle ist eine Tabelle, deren Elemente schwache Referenzen sind. Eine schwache Referenz wird durch die automatische Speicherbereinigung ignoriert. Mit anderen Worten: Falls die einzigen Referenzen auf Objekte schwache Referenzen sind, wird die automatische Speicherbereinigung dieses Objekt einsammeln.
Eine Weak-Tabelle kann schwache Schlüssel, Werte oder beides enthalten. Eine Tabelle mit schwachen Schlüsseln erlaubt das Sammeln dieser, aber bewahrt die Werte vor der Bereinigung. Eine Tabelle mit sowohl schwachen Schlüsseln als auch schwachen Werten erlaubt das Sammeln sowohl der Schlüssel als auch der Werte. In jedem Fall, wenn entweder der Schlüssel oder der Wert gesammelt wird, wird das komplette Paar aus der Tabelle entfernt. Die Schwäche einer Tabelle wird durch das __mode-Feld ihrer Metatabelle gesteuert. Wenn das __mode-Feld eine Zeichenkette mit dem Zeichen 'k' ist, sind die Schlüssel der Tabelle schwach. Wenn __mode 'v' enthält, sind die Werte der

Tabelle schwach.
Nachdem Sie eine Tabelle als Metatabelle verwenden, sollten Sie deren Wert des __mode-Feldes nicht verändern. Andernfalls ist das Referenzverhalten der von dieser Metatabelle kontrollierten Tabellen undefiniert.

2.11 Koroutinen

Lua unterstützt Koroutinen. Eine Koroutine in Lua repräsentiert einen unabhängigen Thread der Ausführung. Im Gegensatz zu Multithread-Systemen hält eine Koroutine ihre Ausführung jedoch nur über den expliziten Aufruf einer "yield"-Funktion an.
Sie erzeugen eine Koroutine über einen Aufruf von coroutine.create. Dessen einziges Argument ist eine Funktion, welche die Hauptfunktion der Koroutine darstellt. Die create-Funktion erzeugt lediglich eine neue Koroutine und liefert einen Verweis auf diese (ein Objekt des Typs thread); sie startet nicht die Ausführung der Koroutine.
Wenn Sie zum ersten mal coroutine.resume aufrufen und als erstes Argument einen von coroutine.create zurückgegebenen Thread übergeben, startet die Koroutine ihre Ausführung bei der ersten Zeile ihrer Hauptfunktion. Zusätzliche Argumente, welche coroutine.resume übergeben werden, werden der Hauptfunktion der Koroutine durchgereicht. Nachdem die Koroutine gestartet ist, läuft diese, bis sie entweder beendet oder unterbrochen wird.
Eine Koroutine kann ihre Ausführung auf zwei Arten beenden: Normalerweise, wenn ihre Hauptfunktion beendet ist (explizit, oder implizit nach der letzten Anweisung) und im Sonderfall, wenn ein ungeschützter Fehler auftritt. Im ersten Fall liefert coroutine.resume true und alle Werte, welche von der Hauptfunktion der Koroutine geliefert werden. Im Fehlerfall liefert coroutine.resume false und eine Fehlernachricht.
Eine Koroutine hält durch einen Aufruf von coroutine.yield an. Wenn eine Koroutine anhält, gibt die zugehörige coroutine.resume sofort zurück, auch wenn das Anhalten in einem verschachtelten Funktionsaufruf (d. h. nicht in der Hauptfunktion, sondern einer Funktion, welche direkt oder indirekt durch die Hauptfunktion aufgerufen wurde) stattfand. Im Falle des Anhaltens liefert coroutine.resume ebenfalls true und sämtliche an coroutine.yield übergebenen Werte. Bei der nächsten Wiederaufnahme der Koroutine führt diese ihre Ausführung an der Stelle des Anhaltens mit einem Aufruf von coroutine.yield, welche sämtliche zusätzliche an coroutine.resume übergebenen Argumente zurückliefert, fort.
Wie coroutine.create erzeugt die coroutine.wrap-Funktion ebenfalls eine Koroutine, aber anstatt diese selbst zurückzuliefern, liefert sie eine Funktion, die wenn sie aufgerufen wird, die Koroutine wieder aufnimmt. Jegliche Argumente, welche dieser Funktion übergeben werden, gehen als zusätzliche Argumente an coroutine.resume.
coroutine.wrap liefert alle Werte, welche von coroutine.resume geliefert werden, mit Ausnahme des ersten (der bool'sche Fehlercode). Im Gegensatz zu coroutine.resume fängt coroutine.wrap keine Fehler; jegliche Fehler werden an den Aufrufer gereicht.
Betrachten Sie beispielsweise folgenden Code:

```
function foo(a)
print("foo",a)
return coroutine.yield(2*a)
end
co = coroutine.create(function (a,b)
print("co-body",a,b)
local r = foo(a+1)
print("co-body",r)
local r,s = coroutine.yield(a+b,a-b)
print("co-body",r,s)
return b,"end"
end)
print("main",coroutine.resume(co,1,10))
print("main",coroutine.resume(co,"r"))
print("main",coroutine.resume(co,"x","y"))
print("main",coroutine.resume(co,"x","y"))
```

Wenn Sie diesen ausführen, wird folgende Ausgabe erzeugt:

3 Die Programmierschnittstelle

Dieser Abschnitt beschreibt die C-API für Lua; dies ist die Menge von C-Funktionen, welche dem Hostprogramm zur Kommunikation mit Lua zur Verfügung stehen. Alle API-Funktionen und damit verbundene Typen und Konstanten sind in der Header-Datei lua.h deklariert.

Obwohl wir die Bezeichnung "Funktion" verwenden, wird jede Einheit der API als Makro bereitgestellt. Jedes dieser Makros verwendet dessen Argumente exakt einmal (mit Ausnahme des ersten Arguments, welches immer ein Lua-Status ist) und erzeugen somit keine versteckten Seiteneffekte.

Wie in den meisten C-Bibliotheken prüfen die API-Funktionen von Lua ihre Argumente nicht auf Gültigkeit oder Konsistenz. Sie können dieses Verhalten jedoch ändern, indem sie Lua mit einer passenden Definition des Makros luai_apicheck in der Datei luaconf.h kompilieren.

3.1 Der Stapelspeicher

Lua benutzt einen virtuellen Stapelspeicher, um Werte von und zu C zu übergeben. Jedes Element dieses Stapelspeichers repräsentiert einen Lua-Wert (nil, Zahl, Zeichenkette etc.).

Immer wenn Lua C aufruft, bekommt die aufgerufene Funktion einen neuen Stapelspeicher, welcher unabhängig von vorherigen oder noch aktiven von C-Funktionen ist. Dieser Stapelspeicher enthält initial sämtliche Argumente für die C-Funktion und dort legt die C-Funktion die zurückzugebenden Ergebnisse für den Aufrufer ab. (s. lua_CFunction).

Der Bequemlichkeit wegen folgen die meisten Abfrageoperationen der API keiner strikten Stapelspeicher-Vorgehensweise. Stattdessen können diese über einen index auf beliebige Elemente im Stapelspeicher zeigen: Ein positiver Index repräsentiert eine absolute Position (beginnend bei 1); ein negativer Index repräsentiert ein Offset, relativ zum oberen Ende des Stapelspeichers. Genauer gesagt, wenn der Stapelspeicher n Elemente besitzt, dann repräsentiert der Index 1 das erste Element (d. h. das Element, welches zuerst auf dem Stapelspeicher abgelegt wurde) und der Index n repräsentiert das letzte Element; der Index -1 repräsentiert ebenfalls das letzte Element (d. h. das Element ganz oben) und der Index -n repräsentiert das erste Element. Wir bezeichnen einen Index als valide wenn dieser zwischen 1 und dem oberen Ende des Stapelspeichers liegt (d. h. wenn 1 = abs(index) = top).

3.2 Stapelspeicher-Größe

Wenn Sie mit der Lua-API interagieren, sind Sie für die Sicherstellung der Konsistenz verantwortlich. Genauer gesagt sind Sie für die Kontrolle von Pufferüberläufen verantwortlich. Sie können die Funktion lua_checkstack zur Vergrößerung des Stapelspeichers verwenden.

Immer, wenn Lua C aufruft, stellt es sicher, dass zumindest LUA_MINSTACK Positionen verfügbar sind. LUA_MINSTACK ist als 20 definiert, so dass Sie sich für gewöhnlich nicht um den Platz sorgen müssen, sofern Ihr Code keine Schleifen enthält, die Elemente auf dem Stapelspeicher ablegen.

Die meisten Abfragefunktionen akzeptieren als Index jeden Wert innerhalb des zur Verfügung stehenden Platzes auf dem Stapelspeicher, d. h. Indizes bis zur maximalen Stapelspeicher-Größe, welchen Sie per lua_checkstack gesetzt haben. Derartige Indizes werden akzeptable Indizes genannt. Formal definieren wir einen akzeptablen Index wie folgt:

(index 0 && index

Beachten Sie, dass 0 niemals ein akzeptabler Index ist.

3.3 Pseudo-Indizes

Sofern nicht anders angegeben, kann jede Funktion, welche einen gültigen Index akzeptiert, ebenso mit einem Pseudo-Index aufgerufen werden, welcher div. Lua-Werte repräsentiert, welche durch C-Code zugreifbar sind, sich jedoch nicht auf dem Stapelspeicher befinden. Pseudo-Indizes werden benutzt, um auf die Thread-Umgebung, die Funktionsumgebung, die Registry und die freien Variablen einer C-Funktion (s. §?3.4)

zuzugreifen.
Die Thread-Umgebung (in welcher sich globale Variablen befinden) ist immer beim Pseudo-Index LUA_GLOBALSINDEX. Die Umgebung der laufenden C-Funktion ist immer beim Pseudo-Index LUA_ENVIRONINDEX.
Um auf den Wert einer globalen Variable zuzugreifen oder diesen zu ändern, können Sie gewöhnliche Tabellen-Operationen auf eine Umgebungstabelle anwenden. Konkret können Sie auf den Wert einer globalen Variablen wie folgt zugreifen:

3.4 C-Closures

Wenn eine C-Funktion erstellt wird, ist es möglich, Werte mit dieser zu verknüpfen – also C-Closures zu erstellen; diese Werte werden freie Variablen genannt und sind für die Funktion bei jedem Aufruf zugreifbar. (s. lua_pushcclosure).
Immer, wenn eine C-Funktion aufgerufen wird, werden ihre freien Variablen an einem spezifischen Pseudo-Index hinterlegt. Diese Pseudo-Indizes werden durch das Makro lua_upvalueindex erzeugt. Der erste Wert, welcher mit einer Funktion verknüpft wird, ist an der Position lua_upvalueindex(1) usw. Jeder Zugriff auf lua_upvalueindex(n) – wobei n größer als die Anzahl der freien Variablen der aktuellen Funktion (aber nicht größer als 256) ist – liefert einen akzeptablen (aber ungültigen) Index.

3.5 Registry

Lua stellt eine Registry zur Verfügung, welche eine vordefinierte Tabelle ist und von C-Code benutzt werden kann, um jegliche Lua-Werte zu speichern, welche gespeichert werden sollen. Diese Tabelle befindet sich immer am Pseudo-Index LUA_REGISTRYINDEX. Jede C-Bibliothek kann Daten in diese Tabelle speichern, sollte jedoch zur Vermeidung von Kollisionen darauf achten, andere Schlüssel als die von anderen Bibliotheken genutzten zu verwenden. Typischerweise sollten Sie als Schlüssel in Ihrem Code eine Zeichenkette verwenden, welche den Namen Ihrer Bibliothek oder Benutzerdaten mit der Adresse eines C-Objects enthält.
Die ganzzahligen Schlüssel der Registry werden vom Referenzierungmechanismus, welcher von der Hilfsbibliothek implementiert wird, verwendet und sollten daher nicht für andere Zwecke verwendet werden.

3.6 Fehlerbehandlung in C

Intern verwendet Lua den longjmp-Mechanismus von C, um Fehler zu behandeln. (Sie können auch Ausnahmen wählen, wenn Sie C++ verwenden; s. Datei luaconf.h.) Wenn Lua mit einem Fehler konfrontiert wird (z. B. Fehler bei der Speicheranforderung, Typenfehler, Syntaxfehler oder Laufzeitfehler) wirft es einen Fehler; d. h. es führt einen sog. "long jump" durch. Eine geschützte Umgebung benutzt setjmp, um einen Wiederherstellungspunkt zu setzen; jeder Fehler springt zum letzten aktiven Wiederherstellungspunkt.
Die meisten Funktionen der API können Fehler werfen, beispielsweise im Falle eines Fehlers bei der Speicheranforderung. Die Dokumentation jeder Funktion gibt an, ob diese einen Fehler werfen kann.
Innerhalb einer C-Funktion können Sie einen Fehler durch das Aufrufen von lua_error werfen.

3.7 Funktionen und Typen

Hier listen wir alle Funktionen und Typen der C-API in alphabetischer Reihenfolge auf. Jede Funktion hat einen Indikator folgender Art:
Das erste Feld, o, gibt an, wie viele Elemente die Funktion vom Stapelspeicher entfernt. Das zweite Feld, p, gibt an, wie viele Elemente die Funktion auf dem Stapelspeicher ablegt. (Jede Funktion legt ihre Ergebnisse nach dem Entfernen der Argumente ab.) Ein Feld der Form x|y bedeutet, dass die Funktion – je nach Situation – x oder y Elemente ablegen (oder entfernen) kann; ein Fragezeichen '?' bedeutet, dass wir nur durch die Argumente nicht ermitteln können, wie viele Elemente die Funktion entfernt/ablegt (z. B. weil sie vom Inhalt des Stapelspeichers abhängen). Das dritte Feld, x, gibt an, ob die Funktion ggf. Fehler wirft: '-' bedeutet, dass die Funktion niemals Fehler wirft; 'm' bedeutet, dass die Funktion nur bei zu wenig Speicher einen Fehler wirft; 'e' bedeutet, dass die Funktion anderweitige Fehler werfen kann; 'v' bedeutet, dass die Funktion

absichtlich einen Fehler wirft.
lua_Alloc
typedef void * (*lua_Alloc) (void *ud,
void *ptr,
size_t osize,
size_t nsize);

lua_atpanic
lua_CFunction lua_atpanic(lua_State *L, lua_CFunction panicf);
Setzt eine neue Panik-Funktion und liefert die alte zurück.
Falls ein Fehler außerhalb einer geschützten Umgebung auftritt, ruft Lua eine Panik-Funktion auf und anschließend exit(EXIT_FAILURE), um die Host-Anwendung zu verlassen. Ihre Panik-Funktion kann diese Beendigung verhindern, indem sie niemals einen Wert liefert (z. B. über einen long jump).
Die Panik-Funktion kann über den Stack auf die Nachricht des Fehlers zugreifen.
lua_call
void lua_call(lua_State *L, int nargs, int nresults);

lua_CFunction
typedef int (*lua_CFunction) (lua_State *L);

lua_checkstack
int lua_checkstack(lua_State *L, int extra);
Stellt sicher, dass es mindestens extra freie Plätze auf dem Stapelspeicher gibt. Liefert false, wenn der Stapelspeicher nicht auf diese Größe wachsen kann. Diese Funktion verkleinert den Stapelspeicher niemals; wenn der Stapelspeicher bereits größer als die neue Größe ist, wird dieser unverändert belassen.
lua_close
void lua_close(lua_State *L);
Zerstört alle gegebenen Objekte des Lua-Status (durch Aufruf der entsprechenden Metamethoden der automatischen Speicherbereinigung, sofern verfügbar) und gibt allen von diesem Status belegten dynamischen Speicher frei. Auf den meisten Plattformen müssen Sie diese Funktion wahrscheinlich nicht aufrufen, weil alle Resourcen nach dem Ende des Host-Programms freigegeben werden. Andererseits können langlaufende Programme wie Daemons oder Webserver es erforderlich machen, nicht mehr benötigte Stati freizugeben sobald diese nicht mehr benötigt werden, um zu vermeiden, dass der Speicherverbrauch dieser Programme zu stark anwächst.
lua_concat
void lua_concat(lua_State *L, int n);
Konkateniert die n oberen Werte auf dem Stapelspeicher, entfernt diese und belässt das Ergebnis oben auf dem Stapelspeicher. Wenn n 1 ist, ist das Ergebnis der einfache Wert auf dem Stapelspeicher (d. h. die Funktion macht nichts); falls n 0 ist, ist das Ergebnis die leere Zeichenkette. Die Konkatenierung wird unser Berücksichtigung der gewöhnlichen Semantik von Lua durchgeführt (s. §?2.5.4).
lua_cpcall
int lua_cpcall(lua_State *L, lua_CFunction func, void *ud);

lua_createtable
void lua_createtable(lua_State *L, int narr, int nrec);
Erzeugt eine neue leere Tabelle und legt sie auf dem Stapelspeicher ab. Der Tabelle wird Speicher für narr Feld-Elemente und nrec Nicht-Feld-Elemente reserviert. Diese Reservierung ist nützlich, wenn Sie genau wissen, wie viele Elemente die Tabelle haben wird. Andernfalls können Sie die Funktion lua_newtable verwenden.
lua_dump
int lua_dump(lua_State *L, lua_Writer writer, void *data);
Liefert einen binären Dump einer Funktion. Diese Funktion erhält eine Lua-Funktion auf dem Stack und liefert binäre Daten, die wenn sie erneut geladen werden, in einer äquivalenten Funktion wie der alten resultieren. Da diese Funktion Teile der Binärdaten

erzeugt, ruft lua_dump die Funktion writer (s. lua_Writer) mit den zu schreibenden Daten auf.
Der zurückgelieferte Wert entspricht dem Fehler des letzten Aufrufes an writer; 0 bedeutet keine Fehler.
Diese Funktion entfernt die Lua-Funktion nicht vom Stack.
lua_equal
int lua_equal(lua_State *L, int index1, int index2);
Liefert 1, wenn die zwei Werte an den gültigen Indizes index1 und index2 nach der Semantik des Lua-Operators == gleich sind (d. h. es werden ggf. Metamethoden aufgerufen). Andernfalls wird 0 geliefert. Die Funktion liefert ebenso 0, wenn einer der Indizes nicht gültig ist.
lua_error
int lua_error(lua_State *L);
Erzeugt einen Lua-Fehler. Die Fehlermeldung (welche ein Lua-Wert beliebigen Typs sein kann) muss sich oben auf dem Stapelspeicher befinden. Diese Funktion führt einen "long jump" durch und liefert somit kein Ergebnis (s. luaL_error).
lua_gc
int lua_gc(lua_State *L, int what, int data);
Steuert die automatische Speicherbereinigung.
Diese Funktion führt verschiedene Aufgaben aus, entsprechend des Wertes des Parameters what:

LUA_GCSTOP: Hält die automatische Speicherbereinigung an.
LUA_GCRESTART: Startet die automatische Speicherbereinigung neu.
LUA_GCCOLLECT: Führt einen vollständigen Lauf der automatischen Speicherbereinigung durch.
LUA_GCCOUNT: Liefert die gegenwärtig von Lua genutzte Menge an Speicher (in KiB).
LUA_GCCOUNTB: Liefert den Rest der Division des aktuellen Speicherverbrauchs durch 1024.
LUA_GCSTEP: Führt einen inkrementellen Schritt der automatischen Speicherbereinigung aus. Die "Schrittweite" wird in nicht spezifizierter Weise durch data bestimmt (größere Werte bedeuten mehr Schritte). Falls Sie die Schrittweite beeinflussen möchten, müssen Sie den Wert von data experimentell bestimmen. Die Funktion liefert 1, wenn der Schritt einen Lauf beendet hat.
LUA_GCSETPAUSE: Setzt data als neuen Wert für die Pause der automatischen Speicherbereinigung (s. §2.10). Die Funktion liefert den vorherigen Wert der Pause.
LUA_GCSETSTEPMUL: Setzt data als neuen Wert des Schrittmultiplikators (s. §2.10). Die Funktion liefert den vorherigen Wert des Schrittmultiplikators.

lua_getallocf
lua_Alloc lua_getallocf (lua_State *L, void **ud);
Liefert die Speicherallokationsfuntion des gegebenen Status. Wenn ud nicht NULL ist, speichert Lua den an lua_newstate übergebenen Zeiger unter *ud.
lua_getfenv
void lua_getfenv(lua_State *L, int index);
Legt die Umgebungstabelle des Wertes am gegebenen Index auf dem Stapelspeicher ab.
lua_getfield
void lua_getfield(lua_State *L, int index, const char *k);
Legt den Wert t[k] auf dem Stapelspeicher ab, wobei t der Wert am gegebenen gültigen Index ist. Wie in Lua kann diese Funktion eine Metamethode für das "index"-Ereignis auslösen (s. §?2.8).
lua_getglobal
void lua_getglobal(lua_State *L, const char *name);
Legt den Wert des globalen name auf dem Stapelspeicher ab. Die Funktion ist als Makro definiert:
#define lua_getglobal(L,s) lua_getfield(L,LUA_GLOBALSINDEX,s)
lua_getmetatable
int lua_getmetatable(lua_State *L, int index);

Legt die Metatabelle des Wertes am gegebenen gültigen Index auf dem Stapelspeicher ab. Falls der Index ungültig ist oder der Wert keine Metatabelle besitzt, liefert die Funktion 0 und legt nichts auf dem Stapelspeicher ab.
lua_gettable
void lua_gettable(lua_State *L, int index);
Legt auf dem Stack den Wert t[k] ab, wobei t der Wert am gegebenen gültigen Index ist und k den Wert auf dem Stack darstellt.
Diese Funktion entfernt den Schlüssel vom Stack. Wie in Lua kann diese Funktion eine Meta-Methode für das "index"-Ereignis aufrufen.
lua_gettop
int lua_gettop(lua_State *L);
Liefert den Index des obersten Elements auf dem Stapelspeicher. Auf Grund der Tatsache, dass Indizes bei 1 beginnen, ist dieses Ergebnis gleich zur Anzahl der Elemente im Stapelspeicher (und somit entspricht 0 einem leeren Stapelspeicher).
lua_insert
void lua_insert(lua_State *L, int index);
Moves the top element into the given valid index, shifting up the elements above this index to open space. Cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.
lua_Integer
typedef ptrdiff_t lua_Integer;
Der Typ, welcher von der Lua-API zur Repräsentation eines ganzzahligen Wertes dient. Standardmäßig handelt es sich um einen ptrdiff_t, welcher für gewöhnlich der größte vorzeichenbehaftete Ganzzahl-Typ ist, welcher von dem Rechner verarbeitet wird.
lua_isboolean
int lua_isboolean(lua_State *L, int index);
Returns 1 if the value at the given acceptable index has type boolean, and 0 otherwise.
lua_iscfunction
int lua_iscfunction(lua_State *L, int index);
Returns 1 if the value at the given acceptable index is a C function, and 0 otherwise.
lua_isfunction
int lua_isfunction(lua_State *L, int index);
Returns 1 if the value at the given acceptable index is a function (either C or Lua), and 0 otherwise.
lua_islightuserdata
int lua_islightuserdata(lua_State *L, int index);
Returns 1 if the value at the given acceptable index is a light userdata, and 0 otherwise.
lua_isnil
int lua_isnil(lua_State *L, int index);
Liefert 1, wenn der Wert am gegebenen Index nil ist und ansonsten 0.
lua_isnone
int lua_isnone(lua_State *L, int index);
Returns 1 if the given acceptable index is not valid (that is, it refers to an element outside the current stack), and 0 otherwise.
lua_isnoneornil
int lua_isnoneornil(lua_State *L, int index);
Returns 1 if the given acceptable index is not valid (that is, it refers to an element outside the current stack) or if the value at this index is nil, and 0 otherwise.
lua_isnumber
int lua_isnumber(lua_State *L, int index);
Liefert 1, wenn der Wert am gegebenen Index eine Zahl oder zu einer solchen konvertierbaren Zeichenkette ist und ansonsten 0
lua_isstring
int lua_isstring(lua_State *L, int index);
Liefert 1, wenn der Wert am gegebenen Index eine Zeichenkette oder Zahl (welche immer zu einer Zeichenkette konvertierbar ist) ist und ansonsten 0.
lua_istable
int lua_istable(lua_State *L, int index);

Liefert 1, wenn der Wert am gegebenen Index eine Tabelle ist und ansonsten 0.
lua_isthread
int lua_isthread(lua_State *L, int index);
Liefert 1, wenn der Wert am gegebenen Index ein Thread ist und ansonsten 0.
lua_isuserdata
int lua_isuserdata(lua_State *L, int index);
Liefert 1, wenn der Wert am gegebenen Index Benutzerdaten sind und ansonsten 0.
lua_lessthan
int lua_lessthan(lua_State *L, int index1, int index2);
Returns 1 if the value at acceptable index index1 is smaller than the value at acceptable index index2, following the semantics of the Lua < operator (that is, may call metamethods). Otherwise returns 0. Also returns 0 if any of the indices is non valid.
lua_load
int lua_load(lua_State *L,
lua_Reader reader,
void *data,
const char *chunkname);

lua_newstate
lua_State *lua_newstate (lua_Alloc f, void *ud);
Creates a new, independent state. Returns NULL if cannot create the state (due to lack of memory). The argument f is the allocator function; Lua does all memory allocation for this state through this function. The second argument, ud, is an opaque pointer that Lua simply passes to the allocator in every call.
lua_newtable
void lua_newtable(lua_State *L);
Erzeugt eine neue leere Tabelle und legt sie auf dem Stapelspeicher ab. Dies ist äquivalent zu lua_createtable(L,0,0).
lua_newthread
lua_State *lua_newthread (lua_State *L);
Erstellt einen neuen Thread, legt ihn auf dem Stack ab und liefert einen Zeiger auf einen lua_State, der diesen Thread repräsentiert. Der neue von dieser Funktion gelieferte Zustand teilt alle globalen Objekte mit dem ursprünglichen Zustand (so beispielsweise Tabellen), hat jedoch einen unabhängigen Ausführungsstapel.
Es existiert keine Funktion, um Threads explizit zu beenden oder zu zerstören. Threads werden wie jedes andere Lua-Objekt von der automatischen Speicherbereinigung behandelt.
lua_newuserdata
void *lua_newuserdata (lua_State *L, size_t size);

lua_next
int lua_next(lua_State *L, int index);

lua_Number
typedef double lua_Number;

lua_objlen
size_t lua_objlen (lua_State *L, int index);
Returns the "length" of the value at the given acceptable index: for strings, this is the string length; for tables, this is the result of the length operator ('#'); for userdata, this is the size of the block of memory allocated for the userdata; for other values, it is 0.
lua_pcall
int lua_pcall(lua_State *L, int nargs, int nresults, int errfunc);

lua_pop
void lua_pop(lua_State *L, int n);
Entfernt n Elemente aus dem Stapelspeicher.
lua_pushboolean

void lua_pushboolean(lua_State *L, int b);
Legt einen Wahrheitswert b auf dem Stapelspeicher ab.
lua_pushcclosure
void lua_pushcclosure(lua_State *L, lua_CFunction fn, int n);

lua_pushcfunction
void lua_pushcfunction(lua_State *L, lua_CFunction f);

lua_pushfstring
const char *lua_pushfstring(lua_State *L, const char *fmt, ...);

lua_pushinteger
void lua_pushinteger(lua_State *L, lua_Integer n);
Legt eine Zahl mit dem Wert n auf dem Stapelspeicher ab.
lua_pushlightuserdata
void lua_pushlightuserdata(lua_State *L, void *p);

lua_pushliteral
void lua_pushliteral(lua_State *L, const char *s);
Dieses Makro ist äquivalent zu lua_pushlstring, kann jedoch nur benutzt werden, wenn s eine literale Zeichenkette ist. In diesen Fällen wird automatisch die Länge der Zeichenkette zur Verfügung gestellt.
lua_pushlstring
void lua_pushlstring(lua_State *L, const char *s, size_t len);
Pushes the string pointed to by s with size len onto the stack. Lua makes (or reuses) an internal copy of the given string, so the memory at s can be freed or reused immediately after the function returns. The string can contain embedded zeros.
lua_pushnil
void lua_pushnil(lua_State *L);
Legt einen Nullwert auf dem Stapelspeicher ab.
lua_pushnumber
void lua_pushnumber(lua_State *L, lua_Number n);
Legt eine Zahl mit dem Wert n auf dem Stapelspeicher ab.
lua_pushstring
void lua_pushstring(lua_State *L, const char *s);
Pushes the zero-terminated string pointed to by s onto the stack. Lua makes (or reuses) an internal copy of the given string, so the memory at s can be freed or reused immediately after the function returns. The string cannot contain embedded zeros; it is assumed to end at the first zero.
lua_pushthread
int lua_pushthread(lua_State *L);
Pushes the thread represented by L onto the stack. Returns 1 if this thread is the main thread of its state.
lua_pushvalue
void lua_pushvalue(lua_State *L, int index);
Legt eine Kopie des Elements am gegebenen Index auf dem Stapelspeicher ab.
lua_pushvfstring
const char *lua_pushvfstring (lua_State *L,
const char *fmt,
va_list argp);
Equivalent to lua_pushfstring, except that it receives a va_list instead of a variable number of arguments.
lua_rawequal
int lua_rawequal(lua_State *L, int index1, int index2);
Returns 1 if the two values in acceptable indices index1 and index2 are primitively equal (that is, without calling metamethods). Otherwise returns 0. Also returns 0 if any of the indices are non valid.
lua_rawget

void lua_rawget(lua_State *L, int index);
Ähnlich wie lua_gettable, führt jedoch einen rohen Zugriff aus (ohne Metamethoden).
lua_rawgeti
void lua_rawgeti(lua_State *L, int index, int n);
Pushes onto the stack the value t[n], where t is the value at the given valid index. The access is raw; that is, it does not invoke metamethods.
lua_rawset
void lua_rawset(lua_State *L, int index);
Ähnlichen wie lua_settable, führt jedoch einen rohen Zugriff aus (ohne Metamethoden).
lua_rawseti
void lua_rawseti(lua_State *L, int index, int n);

lua_Reader
typedef const char * (*lua_Reader) (lua_State *L,
void *data,
size_t *size);

lua_register
void lua_register(lua_State *L,
const char *name,
lua_CFunction f);

lua_remove
void lua_remove(lua_State *L, int index);
Removes the element at the given valid index, shifting down the elements above this index to fill the gap. Cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.
lua_replace
void lua_replace(lua_State *L, int index);
Moves the top element into the given position (and pops it), without shifting any element (therefore replacing the value at the given position).
lua_resume
int lua_resume(lua_State *L, int narg);

lua_setallocf
void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);
Changes the allocator function of a given state to f with user data ud.
lua_setfenv
int lua_setfenv(lua_State *L, int index);
Pops a table from the stack and sets it as the new environment for the value at the given index. If the value at the given index is neither a function nor a thread nor a userdata, lua_setfenv returns 0. Otherwise it returns 1.
lua_setfield
void lua_setfield(lua_State *L, int index, const char *k);

lua_setglobal
void lua_setglobal(lua_State *L, const char *name);
Entfernt einen Wert vom Stapelspeicher und setzt ihn als neuen Wert des globalen name. Die Funktion ist als Makro definiert:
#define lua_setglobal(L,s) lua_setfield(L,LUA_GLOBALSINDEX,s)
lua_setmetatable
int lua_setmetatable(lua_State *L, int index);
Entfernt eine Tabelle vom Stapelspeicher und setzt sie als neue Metatabelle für den Wert am gegebenen Index.
lua_settable
void lua_settable(lua_State *L, int index);

lua_settop

void lua_settop(lua_State *L, int index);
Accepts any acceptable index, or 0, and sets the stack top to this index. If the new top is larger than the old one, then the new elements are filled with nil. If index is 0, then all stack elements are removed.
lua_State
typedef struct lua_State lua_State;

lua_status
int lua_status(lua_State *L);

lua_toboolean
int lua_toboolean(lua_State *L, int index);
Converts the Lua value at the given acceptable index to a C boolean value (0 or 1). Like all tests in Lua, lua_toboolean returns 1 for any Lua value different from false and nil; otherwise it returns 0. It also returns 0 when called with a non-valid index. (If you want to accept only actual boolean values, use lua_isboolean to test the value's type.)
lua_tocfunction
lua_CFunction lua_tocfunction (lua_State *L, int inde
Konvertiert einen Wert am gegebenen Index in eine C-Funktion. Dieser Wert muss eine C-Funktion sein; andernfalls wird NULL geliefert.
lua_tointeger
lua_Integer lua_tointeger (lua_State *L, int index);

lua_tolstring
const char *lua_tolstring (lua_State *L, int index, size_t *len);

lua_tonumber
lua_Number lua_tonumber (lua_State *L, int index);
Konvertiert den Lua-Wert am gegebenen gültigen Index zu einem C-Typen lua_Number (s. lua_Number). Der Lua-Wert muss eine Zahl oder eine zu einer solchen konvertierbaren Zeichenkette sein (s. §?2.2.1);andernfalls liefert lua_tonumber 0.
lua_topointer
const void *lua_topointer (lua_State *L, int index);

lua_tostring
const char *lua_tostring (lua_State *L, int index);
Äquivalent zu lua_tolstring mit len gleich NULL.
lua_tothread
lua_State *lua_tothread (lua_State *L, int index);
Konvertiert den Wert am gegebenen gültigen Index zu einem Lua-Thread (als lua_State* repräsentiert). Dieser Wert muss ein Thread sein; andernfalls liefert die Funktion NULL.
lua_touserdata
void *lua_touserdata (lua_State *L, int index);

lua_type
int lua_type(lua_State *L, int index);
Liefert den Typ des Wertes am gegebenen Index, oder LUA_TNONE für einen ungültigen Index (d. h. ein Index zu einer "leeren" Position auf dem Stapelspeicher). Die von lua_type gelieferten Typen werden durch folgende in der Datei lua.h definierten Konstanten kodiert: LUA_TNIL, LUA_TNUMBER, LUA_TBOOLEAN, LUA_TSTRING, LUA_TTABLE, LUA_TFUNCTION, LUA_TUSERDATA, LUA_TTHREAD und LUA_TLIGHTUSERDATA.
lua_typename
const char *lua_typename (lua_State *L, int tp);
Liefert den Bezeichner des vom Wert tp genutzten Typs, welcher einem der von lua_type gelieferten entsprechen muss.
lua_Writer
typedef int (*lua_Writer) (lua_State *L,

```
const void* p,
size_t sz,
void* ud);
```

lua_xmove
```
void lua_xmove(lua_State *from, lua_State *to, int n);
```

lua_yield
```
int lua_yield(lua_State *L, int nresults);
```

3.8 Die Debug-Schnittstelle

Lua besitzt keine eingebauten Debug-Tools. Stattdessen bietet es eine spezielle Schnittstelle in Form von Funktionen und Hooks. Diese Schnittstelle erlaubt die Konstruktion verschiedener Arten von Debuggen, Profilern und anderen Tools, welche "interne Informationen" des Interpreters benötigen.

lua_Debug
```
typedef struct lua_Debug {
int event;
const char *name; /* (n) */
const char *namewhat; /* (n) */
const char *what; /* (S) */
const char *source; /* (S) */
int currentline; /* (l) */
int nups; /* (u) number of upvalues */
int linedefined; /* (S) */
int lastlinedefined; /* (S) */
char short_src[LUA_IDSIZE]; /* (S) */
/* private part */
other fields
} lua_Debug;
```
Eine Struktur, um verschiedene Informationen über eine aktive Funktion abzulegen. lua_getstack befüllt lediglich den privaten Abschnitt dieser Struktur zur späteren Verwendung. Rufen Sie lua_getinfo auf, um die anderen Felder von lua_Debug mit nützlichen Informationen zu füllen.
Die Felder von lua_Debug haben folgende Bedeutung:

source: Wenn die Funktion durch eine Zeichenkette definiert wurde, ist source diese Zeichenkette. Wenn die Funktion in einer Datei definiert wurde, beginnt source mit einem '@', gefolgt vom Dateinamen.
short_src: Eine "druckbare" Version von source zur Verwendung in Fehlermeldungen.
linedefined: Die Zeilennummer, an der die Definition der Funktion beginnt.
lastlinedefined: Die Zeilennummer, an der die Definition der Funktion endet.
what: Die Zeichenkette "Lua", wenn die Funktion eine Lua-Funktion ist, "C" wenn es eine C-Funktion ist, "main" wenn es der Hauptteil des Chunks ist und "tail", wenn es sich um eine Datei mit Endrekursion handelt. Im letzten Fall besitzt Lua keine weiteren Informationen über die Funktion.
currentline: Die aktuelle Zeile, an der die gegebene Funktion ausgeführt wird. Wenn keine Zeileninformationen verfügbar sind, wird currentline auf -1 gesetzt.
name: Ein passender Name für die Funktion. Weil Funktionen unter Lua Werte erster Ordnung sind, haben diese keinen festen Bezeichner: Manche Funktionen können der Wert mehrerer globaler Variablen sein, während andere lediglich in einem Tabellenfeld gespeichert sind. Die lua_getinfo-Funktion prüft, wie die Funktion aufgerufen wurde, um einen passenden Namen zu finden. Falls kein Name gefunden werden konnte, wird name auf NULL gesetzt.
namewhat: Beschreibt das name-Feld. Der Wert von namewhat kann – entsprechend der aufgerufenen Funktion – "global", "local", "method", "field", "upvalue" oder "" (die leere Zeichenkette) sein. (Lua verwendet die leere Zeichenkette, wenn keine andere Option zuzutreffen scheint.)
nups: Die Anzahl gebundener Variablen der Funktion.

lua_gethook
lua_Hook lua_gethook(lua_State *L);
Liefert die aktuelle Hook-Funktion.

lua_gethookcount
int lua_gethookcount(lua_State *L);
Liefert die aktuelle Anzahl Hooks.

lua_gethookmask
int lua_gethookmask(lua_State *L);
Liefert die aktuelle Hook-Maske.

lua_getinfo
int lua_getinfo(lua_State *L, const char *what, lua_Debug *ar);
Liefert Informationen über eine spezifische Funktion oder Funktionsaufruf.
Um Informationen über einen Funktionsaufruf zu erhalten, muss der Parameter ar ein gültiger Aktivierungsdatensatz, welcher zuvor durch einen Aufruf von lua_getstack gefüllt wurde, sein oder als Argument an einen Hook übergeben werden (s. lua_Hook).
Um Informationen über eine Funktion zu erhalten, legen Sie diese auf dem Stapelspeicher ab und beginnen die Zeichenkette what mit dem Zeichen '>'. (In diesem Fall entfernt lua_getinfo die Funktion vom Stapelspeicher.) Um beispielsweise zu erfahren, in welcher Zeile eine Funktion f definiert wurde, können Sie folgenden Code schreiben:
lua_Debug ar;
lua_getfield(L,LUA_GLOBALSINDEX,"f"); /* globales 'f' erhalten */
lua_getinfo(L,"S",&ar);
printf("%d\n",ar.linedefined);
Jedes Zeichen der Zeichenkette what wählt einige Felder der Struktur ar zur Befüllung aus, oder legt Werte auf dem Stapelspeicher ab:

'n': Befüllt die Felder name und namewhat.
'S': Befüllt die Felder source, short_src, linedefined, lastlinedefined und what.
'l': Befüllt das Feld currentline.
'u': Befüllt das Feld nups.
'f': Legt auf dem Stapelspeicher die auf gegebener Ebene laufende Funktion ab.
'L': Legt auf dem Stapelspeicher eine Tabelle ab, deren Indizes die Nummern der gültigen Zeilen der Funktion sind. (Eine gültige Zeile ist eine Zeile mit zugehörigem Code, d. h. eine Zeile, in welcher Sie einen Haltepunkt setzen können. Nicht-gültige Zeilen beinhalten leere Zeilen und Kommentare.)

Diese Funktion liefert im Fehlerfall 0 (beispielsweise im Falle einer ungültigen Option für what).

lua_getlocal
const char *lua_getlocal(lua_State *L, lua_Debug *ar, int n);
Liefert Informationen über eine lokale Variable eines gegebenen Aktivierungsdatensatzes. Der Parameter ar welcher zuvor durch einen Aufruf von lua_getstack gefüllt wurde, sein oder als Argument an einen Hook übergeben werden (s. lua_Hook). Der Index n wählt die lokale Variable zur Untersuchung aus (1 ist der erste Parameter oder aktive lokale Variable usw., bis hin zur letzten aktiven lokalen Variable). lua_getlocal legt den Wert der Variablen auf dem Stapelspeicher ab und liefert dessen Name.
Variablen-Bezeichner, welche mit '(' (geöffnete Klammer) beginnen repräsentieren interne Variablen (Laufvariablen, temporäre Variablen und lokale C-Funktionsvariablen).
Liefert NULL (und legt nichts ab), wenn der Index größer als die Anzahl aktiver lokaler Variablen ist.

lua_getstack

int lua_getstack(lua_State *L, int level, lua_Debug *ar);
Liefert Informationen über den Laufzeit-Stapelspeicher des Interpreters.
Diese Funktion befüllt Teile einer lua_Debug-Struktur mit einer Identifikation des Aktivierungsdatensatzes der auf gegebener Ebene ausgeführten Funktion. Level 0 ist die aktuell laufende Funktion und Level n+1 ist die Funktion, welche Ebene n aufgerufen hat. Wenn keine Fehler auftreten, liefert lua_getstack 1; wenn sie mit einem höheren Level als die Stapelspeicher-Tiefe aufgerufen wird, liefert sie 0.

lua_getupvalue
const char *lua_getupvalue(lua_State *L, int funcindex, int n);
Liefert Informationen über die gebundenen Variablen eines Closures. (Bei Lua-Funktionen sind gebundene Variablen externe lokale Variablen, welche von der Funktion genutzt werden und konsequenterweise in deren Closure eingebunden werden.) lua_getupvalue erwartet den Index n einer gebundenen Variable, legt deren Wert auf dem Stapelspeicher ab und liefert ihren Namen. funcindex zeigt auf den Closure auf dem Stapelspeicher. (Gebundene Variablen haben keine bestimmte Ordnung, da diese über die ganze Funktion aktiv sind. Deshalb sind diese willkürlich nummeriert.)
Liefert NULL (und legt nichts ab), wenn der Index größer als die Anzahl gebundener Variablen ist. Diese Funktion verwendet für C-Funktionen die leere Zeichenkette "" als Name für alle gebundenen Variablen.

lua_Hook
typedef void (*lua_Hook)(lua_State *L, lua_Debug *ar);
Typ zum Debuggen von Hook-Funktionen.
Wann immer ein Hook aufgerufen wird, wird das Feld event dessen ar-Argument auf das spezifische Ereignis gesetzt, welches den Hook ausgelöst hat. Lua identifiziert diese Ereignisse mit den folgenden Konstanten: LUA_HOOKCALL, LUA_HOOKRET, LUA_HOOKTAILRET, LUA_HOOKLINE und LUA_HOOKCOUNT. Darüber hinaus wird für Zeilen-Ereignisse ebenso das Feld currentline gesetzt. Um den Wert irgendeines anderen Feldes von ar zu erhalten, muss der Hook lua_getinfo aufrufen. Rückgabe-Ereignisse können LUA_HOOKRET (der normale Wert) oder LUA_HOOKTAILRET sein. In letzterem Fall simuliert Lua eine Rückgabe von einer Funktion, welche eine Endrekursion durchführte; in diesem Fall ist es nutzlos, lua_getinfo aufzurufen.
Während Lua Hooks ausführt, unterbindet es andere Aufrufe von Hooks. Falls ein Hook also Lua anweist, eine Funktion oder einen Chunk auszuführen, erfolgt diese Ausführung ohne jegliche Aufrufe von Hooks.

lua_sethook
int lua_sethook(lua_State *L, lua_Hook f, int mask, int count);
Setzt die Hook-Funktion zum Debuggen.
Das Argument f ist die Hook-Funktion. mask gibt an, bei welchen Ereignissen der Hook aufgerufen wird: Die Angabe erfolgt durch ein bitweises ODER der Konstanten LUA_MASKCALL, LUA_MASKRET, LUA_MASKLINE und LUA_MASKCOUNT. Das count-Argument kommt nur zur Geltung, wenn die Maske LUA_MASKCOUNT beinhaltet. Für jedes Ereignis wird der Hook wie folgt aufgerufen:

Der "call"-Hook: wird aufgerufen, wenn der Interpreter eine Funktion aufruft. Der Hook wird direkt nach dem Betreten der Funktion aufgerufen, bevor die Funktion ihre Argumente erhält.
Der "return"-Hook: wird aufgerufen, wenn der Interpreter aus einer Funktion zurückkehrt. Der Hook wird direkt vor dem Verlassen der Funktion aufgerufen. Sie haben keinen Zugriff auf die von der Funktion zurückgelieferten Werte.
Der "line"-Hook: wird aufgerufen, wenn der Interpreter dabei ist, eine neue Zeile Code zu starten, oder wenn er in den Code zurückspringt (auch in die gleiche Zeile). (Dieses Ereignis findet nur statt, während Lua eine Lua-Funktion ausführt.)
Der "count"-Hook: wird aufgerufen, nachdem der Interpreter eine count-Anweisung ausführt. (Dieses Ereignis findet nur statt, während Lua eine Lua-Funktion ausführt.)

Ein Hook wird durch das Setzen von mask auf Null deaktiviert.

lua_setlocal
const char *lua_setlocal(lua_State *L, lua_Debug *ar, int n);
Setzt den Wert einer lokalen Variablen des gegebenen Aktivierungsdatensatzes. Die Parameter ar und n verhalten sich wie bei lua_getlocal (s. lua_getlocal). lua_setlocal weist der Variablen den Wert oben auf dem Stapelspeicher zu und liefert deren Name. Die Funktion entfernt auch den Wert vom Stapelspeicher.
Liefert NULL (und entfernt nichts), wenn der Index größer als die Anzahl aktiver lokaler Variablen ist.

lua_setupvalue
const char *lua_setupvalue(lua_State *L, int funcindex, int n);
Setzt den Wert der gebundenen Variablen eines Closures. Diese Funktion weist den Wert der oben auf dem Stapelspeicher abgelegten gebundenen Variablen zu und liefert deren Namen. Sie entfernt diesen Wert auch vom Stapelspeicher. Die Parameter funcindex und n verhalten sich wie bei lua_getupvalue (s. lua_getupvalue).
Liefert NULL (und entfernt nichts), wenn der Index größer als die Anzahl gebundener Variablen ist.

4 Die Hilfsbibliothek

Die Hilfsbibliothek stellt einige bequeme Funktionen zur Verbindung von C mit Lua zur Verfügung. Während die Basisbibliothek die primitiven Funktionen für alle Interaktionen zwischen C und Lua zur Verfügung stellt, bietet die Hilfsbibliothek höhere Funktionen für einige häufige Aufgaben an.
Alle Funktionen der Hilfsbibliothek sind in der Header-Datei lauxlib.h definiert und haben das Präfix luaL_.
Alle Funktionen der Hilfsbibliothek basieren auf der Basis-API und bieten somit nichts, was nicht auch mit dieser erledigt werden könnte.
Einige Funktionen der Hilfsbibliothek werden genutzt, um C-Funktionsargumente zu prüfen. Deren Bezeichnung ist immer luaL_check* oder luaL_opt*. Alle diese Funktionen werfen einen Fehler, wenn die Prüfung fehlschlägt. Weil die Fehlermeldung für Argumente formatiert wird – z. B. "bad argument #1") – sollten Sie diese Funktionen nicht für andere Werte verwenden.

4.1 Funktionen und Typen

Hier führen wir alle Funktionen und Typen der Hilfsbibliothek in alphabetischer Reihenfolge auf.
luaL_addchar
void luaL_addchar(luaL_Buffer *B, char c);
Fügt das Zeichen c dem Puffer B hinzu (s. luaL_Buffer).
luaL_addlstring
void luaL_addlstring(luaL_Buffer *B, const char *s, size_t l);
Fügt die Zeichenkette, auf welche s zeigt, mit der Länge l dem Puffer B hinzu (s. luaL_Buffer). Die Zeichenkette darf eingebettete Nullen enthalten.
luaL_addsize
void luaL_addsize(luaL_Buffer *B, size_t n);
Fügt dem Puffer B (s. luaL_Buffer) eine Zeichenkette der Länge n hinzu, welche zuvor in den Pufferbereich kopiert wurde (s. luaL_prepbuffer).
luaL_addstring
void luaL_addstring(luaL_Buffer *B, const char *s);
Fügt die nullterminierte Zeichenkette, auf welche s zeigt, dem Puffer B hinzu (s. luaL_Buffer). Die Zeichenkette darf keine eingebetteten Nullen enthalten.
luaL_addvalue
void luaL_addvalue(luaL_Buffer *B);
Legt den Wert auf dem Stapelspeicher des Puffers von B ab (s. luaL_Buffer). Entfernt den Wert.
Dies ist die einzige auf Zeichenkettenpuffer-bezogene Funktion, welche mit einem zusätzlichen Element auf dem Stapelspeicher aufgerufen werden kann (und muss),

welches jenes ist, welches dem Puffer hinzugefügt wird.
luaL_argcheck
void luaL_argcheck (lua_State *L,
int cond,
int narg,
const char *extramsg);
Prüft, ob cond wahr ist. Wenn dies nicht der Fall ist, wird ein Fehler mit folgender Nachricht geworfen, wobei func vom Aufrufstapel stammt:
bad argument # to ()
luaL_argerror
int luaL_argerror(lua_State *L, int narg, const char *extramsg);
Wirft einen Fehler mit folgender Nachricht, wobei func vom Aufrufstapel kommt:
bad argument # to ()
Diese Funktion liefert nichts zurück; es ist ein Idiom zur Verwendung in C-Funktionen als return luaL_argerror(args).
luaL_Buffer
typedef struct luaL_Buffer luaL_Buffer;
Typ für einen Zeichenketten-Puffer.
Ein Zeichenketten-Puffer erlaubt C-Code, Lua-Zeichenketten stückweise zu generieren. Das Vorgehensmuster ist dabei wie folgt:

Zuerst deklarieren Sie eine Variable b des Typs luaL_Buffer.
Dann initialisieren Sie diese mit einem Aufruf von luaL_buffinit(L,&b).
Anschließend fügen Sie Zeichenketten durch einen Aufruf der luaL_add*-Funktionen dem Puffer hinzu.
Abschließend stellen Sie dies durch einen Aufruf von luaL_pushresult(&b) fertig. Dieser Aufruf belässt die finale Zeichenkette auf dem Stapelspeicher.

Während der normalen Verwendung benutzt ein Zeichenkette-Puffer eine variable Anzahl von Plätzen im Stapelspeicher. Während der Verwendung eines Puffers können Sie somit nicht davon ausgehen, die obere Position des Stapelspeichers zu kennen. Sie können den Stapelspeicher zwischen sukzessiven Aufrufen der Puffer-Funktionen verwenden, sofern diese Verwendung balanciert ist; d. h. wenn Sie eine Puffer-Operation aufrufen, ist der Stapelspeicher auf gleicher Ebene, wie er unmittelbar nach der vorherigen Puffer-Operation war. (Die einzige Ausnahme zu dieser Regel ist luaL_addvalue.) Nach dem Aufruf von luaL_pushresult besitzt der Stapelspeicher die gleiche Höhe wie zum Zeitpunkt als der Puffer initialisiert wurde, plus die finale Zeichenkette ganz oben.
luaL_buffinit
void luaL_buffinit(lua_State *L, luaL_Buffer *B);
Initialisiert einen Puffer B. Diese Funktion alloziert keinen Speicher; der Puffer muss als Variable deklariert werden (s. luaL_Buffer).
luaL_callmeta
int luaL_callmeta(lua_State *L, int obj, const char *e);
Ruft eine Metamethode auf.
Wenn das Objekt beim Index obj eine Metatabelle besitzt und diese das Feld e, ruft diese Funktion dieses Feld auf und übergibt das Objekt als einziges Argument. In diesem Fall liefert die Funktion 1 und legt den vom Aufruf zurückgegebenen Wert auf dem Stapelspeicher ab. Falls keine Metatabelle oder Metamethode existiert, liefert diese Funktion 0 (ohne einen Wert auf dem Stapelspeicher abzulegen).
luaL_checkany
void luaL_checkany(lua_State *L, int narg);
Prüft, ob die Funktion ein Argument beliebigen Typs (inkl. nil) an Position narg besitzt.
luaL_checkint
int luaL_checkint(lua_State *L, int narg);
Prüft, ob das Funktionsargument narg eine Zahl ist und liefert diese zu int konvertiert.
luaL_checkinteger
lua_Integer luaL_checkinteger(lua_State *L, int narg);
Prüft, ob das Funktionsargument narg eine Zahl ist und liefert diese zu einem lua_Integer

konvertiert.
luaL_checklong
long luaL_checklong(lua_State *L, int narg);
Prüft, ob das Funktionsargument narg eine Zahl ist und liefert diese zu long konvertiert.
luaL_checklstring
const char *luaL_checklstring(lua_State *L, int narg, size_t *l);
Prüft, ob das Funktionsargument narg eine Zeichenkette ist und liefert diese; sofern l nicht NULL ist, wird *l mit der Länge der Zeichenkette befüllt.
Diese Funktion verwendet lua_tolstring um ihr Ergebnis zu erhalten, so dass alle Konvertierungen und Hinweise dieser Funktion hier zum Tragen kommen.
luaL_checknumber
lua_Number luaL_checknumber(lua_State *L, int narg);
Prüft, ob das Argument narg eine Zahl ist und liefert diese Zahl.
luaL_checkoption
int luaL_checkoption (lua_State *L,
int narg,
const char *def,
const char *const lst[]);
Prüft, ob das Funktionsargument narg eine Zeichenkette ist und sucht im Feld lst (welches nullterminiert sein muss) nach dieser. Liefert den Index des Feldes, an welchem die Zeichenkette gefunden wurde. Wirft einen Fehler, wenn das Argument keine Zeichenkette war oder diese nicht gefunden werden konnte.
Wenn def nicht NULL ist, verwendet die Funktion def als Standardwert, wenn es kein Argument narg gibt oder dieses nil ist.
Dies ist eine nützliche Funktion zum Mapping von Zeichenketten auf C-Aufzählungen. (Die gewöhnliche Konvention verwendet in Lua-Bibliotheken Zeichenketten statt Nummern zur Auswahl von Optionen.)
luaL_checkstack
void luaL_checkstack(lua_State *L, int sz, const char *msg);
Vergrößert den Stapelspeicher auf top + sz Elemente, wobei ein Fehler geworfen wird, wenn der Stapelspeicher nicht zu dieser Größe anwachsen kann. msg ist ein zusätzlicher Text für die Fehlernachricht.
luaL_checkstring
const char *luaL_checkstring(lua_State *L, int narg);
Prüft, ob das Funktionsargument narg eine Zeichenkette ist und liefert diese.
Diese Funktion verwendet lua_tolstring um ihr Ergebnis zu erhalten, so dass alle Konvertierungen und Hinweise dieser Funktion hier zum Tragen kommen.
luaL_checktype
void luaL_checktype(lua_State *L, int narg, int t);
Prüft, ob das Funktionsargument narg vom Typ t ist. Siehe lua_type für die Kodierung der Typen für t.
luaL_checkudata
void *luaL_checkudata(lua_State *L, int narg, const char *tname);
Prüft, ob das Funktionsargument narg Benutzerdaten des Typs tname sind (s. luaL_newmetatable).
luaL_dofile
int luaL_dofile(lua_State *L, const char *filename);
Lädt die gegebene Datei und führt diese aus. Sie wird durch folgendes Makro definiert:
(luaL_loadfile(L, filename) || lua_pcall(L, 0, LUA_MULTRET, 0))
Liefert im Erfolgsfall 0, oder 1 wenn Fehler auftreten.
luaL_dostring
int luaL_dostring(lua_State *L, const char *str);
Lädt die gegebene Zeichenkette und führt diese aus. Sie wird durch folgendes Makro definiert:
(luaL_loadstring(L, str) || lua_pcall(L, 0, LUA_MULTRET, 0))
Liefert im Erfolgsfall 0, oder 1 wenn Fehler auftreten.
luaL_error
int luaL_error(lua_State *L, const char *fmt, ...);

Wirft einen Fehler. Das Format der Fehlernachricht wird durch fmt und alle zusätzlichen Argumente definiert, wobei nach der gleichen Vorgehensweise wie bei lua_pushfstring verfahren wird. Am Anfang der Nachricht wird außerdem der Dateiname und ggf. die Fehlerzeile hinzugefügt, sofern diese Informationen verfügbar sind.
Diese Funktion liefert nichts zurück; es ist ein Idiom zur Verwendung in C-Funktionen als return luaL_error(args).
luaL_getmetafield
int luaL_getmetafield (lua_State *L, int obj, const char *e);
Legt das Feld e der Metatabelle des Objekts am Index obj auf dem Stapelspeicher ab. Wenn das Objekt keine Metatabelle besitzt oder die Metatabelle das Feld nicht besitzt, wird 0 geliefert und nichts abgelegt.
luaL_getmetatable
void luaL_getmetatable (lua_State *L, const char *tname);
Legt die mit dem Namen tname assoziierte Metatabelle aus der Registry auf dem Stapelspeicher ab (s. luaL_newmetatable).
luaL_gsub
const char *luaL_gsub (lua_State *L,
const char *s,
const char *p,
const char *r);
Erzeugt eine Kopie der Zeichenkette s, wobei jedes Auftreten der Zeichenkette p durch die Zeichenkette r ersetzt wird. Legt die sich ergebende Zeichenkette auf dem Stapelspeicher ab und liefert sie zurück.
luaL_loadbuffer
int luaL_loadbuffer (lua_State *L,
const char *buff,
size_t sz,
const char *name);
Lädt einen Puffer als Lua-Code. Diese Funktion verwendet lua_load, um den Code aus dem Puffer, auf welchen von buff mit der Größe sz gezeigt wird, zu laden.
Diese Funktion liefert die gleichen Ergebnisse wie lua_load. name ist der Bezeichner, welcher für Debug-Informationen und Fehlermeldungen genutzt wird.
luaL_loadfile
int luaL_loadfile (lua_State *L, const char *filename);
Lädt eine Datei als Lua-Code. Diese Funktion verwendet lua_load, um den Code in die Datei namens filename zu laden. Wenn filename NULL ist, wird von der Standardeingabe geladen. Die erste Zeile der Datei wird ignoriert, wenn diese mit # beginnt.
Diese Funktion liefert die gleichen Ergebnisse wie lua_load, besitzt jedoch einen zusätzlichen Fehlercode LUA_ERRFILE, wenn die Datei nicht geöffnet/geladen werden kann.
Genauso wie lua_load lädt diese Funktion den Code lediglich; er wird nicht ausgeführt.
luaL_loadstring
int luaL_loadfile(lua_State *L, const char *filename);
Lädt eine Zeichenkette als Lua-Code. Diese Funktion verwendet lua_load um den Code in die nullterminierte Zeichenkette s zu laden.
Diese Funktion liefert das gleiche Ergebnis wie lua_load.
Genauso wie lua_load lädt diese Funktion den Code lediglich; er wird nicht ausgeführt.
luaL_newmetatable
int luaL_newmetatable(lua_State *L, const char *tname);
Wenn die Registry bereits den Schlüssel tname enthält, wird 0 geliefert. Andernfalls wird eine neue Tabelle zur Verwendung als Metatabelle für die Benutzerdaten erzeugt, der Registry mit dem Schlüssel tname hinzugefügt und 1 geliefert.
In beiden Fällen wird der finale mit tname in der Registrierung verküpfte Wert auf dem Stapelspeicher abgelegt.
luaL_newstate
lua_State *luaL_newstate(void);
Erstellt einen neuen Lua-Status. Es wird die Funktion lua_newstate mit einem auf der standardmäßigen C-Funktion realloc basierten Allocator aufgerufen und anschließend eine

"panic"-Funktion gesetzt (s. lua_atpanic), welche im Falle eines fatalen Fehlers eine Fehlermeldung auf der Standard-Fehlerausgabe ausgibt.
Liefert den neuen Status oder NULL, wenn ein Fehler bei der Speicheranforderung auftritt.
luaL_openlibs
void luaL_openlibs(lua_State *L);
Öffnet alle Lua-Standardbibliotheken im gegebenen Status.
luaL_optint
int luaL_optint(lua_State *L, int narg, int d);
Wenn das Funktionsargument narg eine Zahl ist, wird diese zu einem int konvertiert geliefert. Falls das Argument nicht angegeben wird oder nil ist, wird d geliefert. Andernfalls wird ein Fehler geworfen.
luaL_optinteger
lua_Integer luaL_optinteger (lua_State *L,
int narg,
lua_Integer d);
Wenn das Funktionsargument narg eine Zahl ist, wird diese zu lua_Integer konvertiert geliefert. Falls das Argument nicht angegeben wird oder nil ist, wird d geliefert. Andernfalls wird ein Fehler geworfen.
luaL_optlong
long luaL_optlong(lua_State *L, int narg, long d);
Wenn das Funktionsargument narg eine Zahl ist, wird diese zu long konvertiert geliefert. Falls das Argument nicht angegeben wird oder nil ist, wird d geliefert. Andernfalls wird ein Fehler geworfen.
luaL_optlstring
const char *luaL_optlstring (lua_State *L,
int narg,
const char *d,
size_t *l);
Wenn das Funktionsargument narg eine Zeichenkette ist, wird diese geliefert. Falls das Argument nicht angegeben wird oder nil ist, wird d geliefert. Andernfalls wird ein Fehler geworfen.
Wenn l nicht NULL ist, wird die Position *l mit der Länge des Ergebnisses befüllt.
luaL_optnumber
lua_Number luaL_optnumber(lua_State *L, int narg, lua_Number d);
Wenn das Funktionsargument narg eine Zahl ist, wird diese geliefert. Falls das Argument nicht angegeben wird oder nil ist, wird d geliefert. Andernfalls wird ein Fehler geworfen.
luaL_optstring
const char *luaL_optstring (lua_State *L,
int narg,
const char *d);
Wenn das Funktionsargument narg eine Zeichenkette ist, wird diese geliefert. Falls das Argument nicht angegeben wird oder nil ist, wird d geliefert. Andernfalls wird ein Fehler geworfen.
luaL_prepbuffer
char *luaL_prepbuffer(luaL_Buffer *B);
Liefert die Adresse zu einem Speicherplatz der Größe LUAL_BUFFERSIZE, wohin Sie eine Zeichenkette kopieren können, welche dem Puffer B hinzugefügt werden soll (s. luaL_Buffer). Nach dem Kopieren der Zeichenkette in diesen Bereich müssen Sie luaL_addsize mit der Länge der Zeichenkette aufrufen, um das eigentliche Hinzufügen zum Puffer durchzuführen.
luaL_pushresult
void luaL_pushresult(luaL_Buffer *B);
Beendet die Verwendung des Puffers B, wobei die verbleibende Zeichenkette auf dem Stapelspeicher belassen wird.
luaL_ref
int luaL_ref(lua_State *L, int t);
Erzeugt und liefert eine Referenz in der Tabelle beim Index t für das Objekt auf dem

Stapelspeicher (und entfernt dieses).
Eine Referenz ist ein eindeutiger, ganzzahliger Schlüssel. Sofern Sie keine ganzzahligen Schlüssel manuell in die Tabelle t hinzufügen, sichert luaL_ref die Eindeutigkeit der gelieferten Schlüssel zu. Sie können ein durch die Referenz r verwiesenes Objekt durch einen Aufruf von lua_rawgeti(L,t,r) erhalten. Die Funktion luaL_unref entfernt eine Referenz und das damit verbundene Objekt.
Falls das Objekt auf dem Stapelspeicher nil ist, liefert luaL_ref die Konstante LUA_REFNIL. Die Konstante LUA_NOREF unterscheidet sich garantiert von jeder von luaL_ref gelieferten Referenz.
luaL_Reg
typedef struct luaL_Reg {
const char *name;
lua_CFunction func;
} luaL_Reg;
Typ für Felder von Funktion, um per luaL_register registriert zu werden. name ist der Funktionsname und func ist ein Zeiger auf die Funktion. Jedes Feld des Typs luaL_Reg muss mit einem Markierungseintrag enden, bei welchem sowohl name als auch func NULL sind.
luaL_register
void luaL_register (lua_State *L,
const char *libname,
const luaL_Reg *l);
Öffnet eine Bibliothek.
Wenn der Aufruf mit NULL als libname erfolgt, werden lediglich alle Funktionen der Liste l (s. luaL_Reg) in der Tabelle auf dem Stapelspeicher registriert.
Wenn der Aufruf mit einem von NULL verschiedenen libname erfolgt, erzeugt luaL_register eine neue Tabelle t, setzt diese als Wert der globalen Variable libname und von package.loaded[libname] und registriert darin alle Funktionen aus der Liste l. Falls sich eine Tabelle unter package.loaded[libname] oder in der Variablen libname befindet, wird diese genutzt anstatt eine neue zu erzeugen.
In jedem Fall belässt die Funktion die Tabelle auf dem Stapelspeicher.
luaL_typename
const char *luaL_typename(lua_State *L, int index);
Liefert den Namen des Typs des Wertes am gegebenen index.
luaL_typerror
int luaL_typerror(lua_State *L, int narg, const char *tname);
Erzeugt einen Fehler mit einer Nachricht wie der Folgenden:
location: bad argument narg to 'func' (tname expected, got rt)
... wobei location durch luaL_where erzeugt wird, func der Bezeichner der aktuellen Funktion ist und rt der Name des Typs des eigentlichen Arguments.
luaL_unref
void luaL_unref(lua_State *L, int t, int ref);
Entfernt die Referenz ref von der Tabelle am Index t (s. luaL_ref). Der Eintrag von der Tabelle wird entfernt, so dass das referenzierte Objekt bereinigt werden kann. Die Referenz ref wird darüber hinaus zur Wiederverwendung freigegeben.
Wenn ref LUA_NOREF oder LUA_REFNIL entspricht, führt luaL_unref keine Aktion durch.
luaL_where
void luaL_where(lua_State *L, int lvl);
Legt auf dem Stapelspeicher eine Zeichenkette ab, welche die aktuelle Position des Ablaufs des Levels lvl auf dem Aufrufstapel identifiziert. Typischerweise hat diese Zeichenkette das folgende Format:
chunkname:currentline:
Level 0 ist die laufende Funktion, Level 1 ist die die laufende Funktion aufrufende Funktion etc.
Diese Funktion wird zur Erzeugung eines Präfix für Fehlermeldungen verwendet.

5 Die Standardbibliotheken

Die Lua-Standardbibliotheken stellen nützliche Funktionen zur Verfügung, welche direkt über die C-API implementiert sind. Manche dieser Funktionen stellen essentielle Dienste für die Sprache zur Verfügung (z. B. type und getmetatable); andere stellen Zugriff auf "externe" Dienste zur Verfügung (z. B. I/O) und andere wiederum könnten auch in Lua selbst implementiert werden, sind jedoch sehr nützlich oder haben kritische Laufzeitanforderungen, welche eine Implementierung in C erforderlich machen (z. B. table.sort).
Alle Bibliotheken sind über die offizielle C-API implementiert und werden als separate C-Module angeboten. Derzeit besitzt Lua die folgenden Standardbibliotheken:

Basisbibliothek, welche die Koroutinen-Unterbibliothek beinhaltet
Paketierungsbibliothek
Zeichenkettenverarbeitung
Tabellenverarbeitung
Mathematische Funktionen (sin, log etc.)
Eingabe und Ausgabe
Betriebssystem-Interaktion
Debug-Möglichkeiten

Mit Ausnahme der Basis- und Paketierungsbibliotheken stellt jede Bibliothek ihre Funktionalität als Felder einer globalen Tabelle oder Methoden ihres Objekts zur Verfügung.
Um Zugriff auf diese Bibliotheken zu erhalten, sollte das C-Hostprogramm die luaL_openlibs-Funktion aufrufen, welche alle Standardbibliotheken öffnet. Alternativ kann es diese individuell durch einen Aufruf von luaopen_base (für die Basisbibliothek), luaopen_package (für die Paketierungsbibliothek), luaopen_string (für die Zeichenketten-Bibliothek), luaopen_table (für die Tabellenbibliothek), luaopen_math (für die mathematische Bibliothek), luaopen_io (für die Ein-/Ausgabebibliothek), luaopen_os (für die Betriebssystem-Bibliothek) oder luaopen_debug (für die Debug-Bibliothek) öffnen. Diese Funktionen sind in der Datei lualib.h deklariert und sollten nicht direkt aufgerufen werden: Sie müssen diese wie jede andere C-Funktion von Lua aufrufen, z. B. durch Benutzung von lua_call.

5.1 Basisfunktionen

Die Basisbibliothek bietet einige grundsätzliche Funktionen für Lua. Wenn Sie diese Bibliothek nicht in Ihre Anwendung einbinden, sollten Sie sorgfältig prüfen, ob Sie für einige der Funktionen Implementierungen anbieten müssen.
assert(v[,message])
Verursacht einen Fehler, wenn der Wert des Arguments v falsch (also nil oder false) ist; andernfalls werden alle Argumente zurückgeliefert. message ist eine Fehlermeldung; wenn nicht angegeben lautet der Standard "assertion failed!"
collectgarbage(opt[,arg])
Diese Funktion ist eine generische Schnittstelle zur automatischen Speicherbereinigung. Sie führt div. Funktionen entsprechend ihres ersten Arguments opt durch:

"stop": Hält die automatische Speicherbereinigung an.
"restart": Startet die automatische Speicherbereinigung neu.
"collect": Führt einen vollständigen Zyklus der automatischen Speicherbereinigung durch.

"count": Liefert den gesamten von Lua genutzten Speicher (in KB)
"step": Führt einen Schritt der automatischen Speicherbereinigung durch. Die "Schrittweite" wird durch arg in nicht spezifizierter Weise (größere Werte bedeuten mehr Schritte) gesteuert. Falls Sie die Schrittweite steuern möchte, müssen Sie den Wert von arg experimentell feinjustieren. Liefert true, wenn der Schritt einen Sammelzyklus beendet hat.
"setpause": Setzt arg als neuen Wert für die Pause der automatischen Speicherbereinigung (s. §?2.10).Liefert den vorherigen Wert der Pause.
"setstepmul": Setzt arg als neuen Wert für step multiplier der automatischen Speicherbereinigung (s. §?2.10).Liefert den vorherigen Wert der Schrittweite.

dofile(filename)
Öffnet die benannte Datei und führt deren Inhalt als Lua-Code aus. Bei Aufruf ohne Argumente führt dofile den Inhalt der Standardeingabe (stdin) aus. Gibt alle vom Code gelieferten Werte zurück. Im Fehlerfall reicht dofile diesen an den Aufrufer weiter (d. h. dofile läuft nicht im geschützen Modus).
error(message[,level])
Beendet die zuletzt aufgerufene geschützte Funktion und liefert message als Fehlermeldung. Die Funktion error selbst liefert keinen Wert.
Für gewöhnlich fügt error am Anfang der Nachricht einige Informationen über die Fehlerposition hinzu. Das level-Argument gibt an, wie die Fehlerposition ermittelt werden soll. Mit Level 1 (Standard) ist die Fehlerposition dort, von wo aus die error-Funktion aufgerufen wurde. Level 2 gibt den Fehler dort an, von wo aus die Funktion, welche error aufgerufen hat, aufgerufen wurde usw. Das Übergeben eines Level 0 unterbindet das Hinzufügen von Fehlerpositions-Informationen zur Nachricht.
_G
Eine globale Variable (keine Funktion), welche die globale Umgebung enthält (d. h. _G._G = _G). Lua selbst verwendet diese Variable nicht; das Ändern des Wertes beeinfluss keine Umgebung. (Verwenden Sie setfenv zum Ändern von Umgebungen.)
getfenv([f])
Liefert die derzeit von der Funktion genutzte Umgebung. f kann eine Lua-Funktion sein, oder aber eine Nummer, welche die Funktion der entsprechenden Ebene des Stapelspeichers angibt: Level 1 ist die Funktion, welche getfenv aufruft. Wenn die gegebene Funktion keine Lua-Funktion oder f 0 ist, liefert getfenv die globale Umgebung. Der Standard für f ist 1.
getmetatable(object)
Wenn object keine Metatabelle enthält, wird nil geliefert. Wenn die Metatabelle des Objekts ein "__metatable"-Feld enthält, wird der zugeordnete Wert geliefert. Andernfalls wird die Metatabelle des gegebenen Objekts geliefert.
ipairs(t)
Liefert drei Werte: Eine Iterator-Funktion, die Tabelle t und 0, so dass die Konstruktion …

for i,v in ipairs(t) do body end
… über die Paare (1,t[1]), (2,t[2]), ⋯ bis zum ersten nicht mehr in der Tabelle enthaltenen ganzzahligen Schlüssel iterieren wird.
load(func[,chunkname])
Lädt einen Code mit Hilfe der Funktion func, um dessen Teile zu erhalten. Jeder Aufruf von func muss eine mit vorherigen Ergebnissen verknüpfte Zeichenkette sein. Die Rückgabe von einer leeren Zeichenkette, nil oder keinem Wert signalisiert das Ende des Codes.
Wenn keine Fehler auftreten, wird der kompilierte Code als Funktion geliefert; andernfalls wird nil und die Fehlernachricht geliefert. Die Umgebung der zurückgelieferten Funktion ist die globale Umgebung.
chunkname wird als Bezeichner des Codes für Fehlernachrichten und Debug-Informationen verwendet. Wird dies nicht angegeben, fällt er auf "=(load)" zurück.
loadfile([filename])
Ähnlich wie load, erhält den Code jedoch aus der Datei filename oder von der Standardeingabe, wenn kein Dateiname angegeben wurde.
loadstring(string[,chunkname])
Ähnlich wie load, erhält den Code jedoch aus der übergebenen Zeichenkette.
Um eine gegebene Zeichenkette zu laden und auszuführen verwenden Sie folgende Schreibweise:
assert(loadstring(s))()
Wird chunkname nicht angegebenen, wird auf die gegebene Zeichenkette zurückgefallen.

next(table[,index])
Ermöglicht einem Programm, alle Felder einer Tabelle zu durchlaufen. Das erste Argument ist eine Tabelle und das zweite ein Index dieser Tabelle. next liefert den

nächsten Index dieser Tabelle und dessen verknüpften Wert. Bei einem Aufruf mit nil als zweitem Argument liefert next einen initialen Index und dessen verknüpften Wert. Erfolgt ein Aufruf mit dem letzten Index oder mit nil in einer leeren Tabelle, liefert next nil. Wird das zweite Argument nicht angegeben, wird dieses als nil interpretiert. Im Speziellen können Sie next(t) verwenden, um zu überprüfen, ob eine Tabelle leer ist.
Die Richtung, in welcher die Indizes sortiert werden, ist nicht spezifiziert – auch für numerische Indizes. (Um eine Tabelle in numerischer Sortierung zu traversieren, verwenden Sie das numerische for oder die ipairs-Funktion.)
The behavior of next is undefined if, during the traversal, you assign any value to a non-existent field in the table. Sie können jedoch bestehende Felder modifizieren. Im Speziellen können Sie bestehende Felder leeren.
pairs(t)
Liefert drei Werte: Die next-Funktion, die Tabelle t und nil, so dass die Konstruktion …
for k,v in pairs(t) do body end
… über alle Schlüssel/Werte-Paare der Tabelle t iterieren wird.
S. die Funktion next für die Warnungen zur Modifikation von Tabellen während deren Traversierung.
pcall(f,arg1,···)
Ruft die Funktion f mit den gegebenen Argumenten im geschützten Modus auf. Das bedeutet, dass ein Fehler in f nicht weitergereicht wird; stattdessen fängt pcall den Fehler und liefert einen Statuscode. Das erste Ergebnis ist der Statuscode (ein Wahrheitswert), welcher true ist, wenn der Aufruf ohne Fehler erfolgte. In so einem Fall liefert pcall zusätzlich alle Ergebnisse des Aufrufs nach diesem ersten Ergebnis. Im Falle eines Fehlers liefert pcall false und die Fehlermeldung.
print(···)
Nimmt eine beliebige Anzahl Argumente entgegen und gibt deren Werte unter Zuhilfenahme der tostring-Funktion zu deren Konvertierung zu Zeichenketten auf stdout aus. print ist nicht für formatierte Ausgaben gedacht, sondern lediglich als schnelle Möglichkeit, Werte auszugeben – typischerweise zum Debugging. Für formatierte Ausgaben verwenden Sie string.format.
rawequal(v1,v2)
Prüft, ob v1 gleich zu v2 ist, ohne dabei Metamethoden aufzurufen. Liefert einen Wahrheitswert.
rawget(table,index)
Liefert den Wert von table[index], ohne Metamethoden aufzurufen. table muss eine Tabelle sein; index kann ein beliebiger Wert sein.
rawset(table,index,value)
Setzt den Wert von table[index] auf value, ohne dabei Metamethoden aufzurufen. table muss eine Tabelle sein, index ein von nil verschiedener Wert und value ein Lua-Wert. Diese Funktion liefert table.
select(index,···)
Wenn index eine Zahl ist, werden alle Argumente nach dem Argument index geliefert. Andernfalls muss index die Zeichenkette "#" sein, damit select die Gesamtzahl der zusätzlich übergebenen Argumente zu liefert.
setfenv(f,table)
Setzt die von der gegebenen Funktion zu nutzende Umgebung. f kann eine Lua-Funktion sein, oder eine Zahl, welche die Funktion an entsprechender Stelle im Stapelspeicher angibt: Level 1 ist die Funktion, welche setfenv aufruft. setfenv liefert die gegebene Funktion.
Als Spezialfall ändert setfenv die Umgebung des laufenden Threads, wenn f 0 ist. In diesem Fall liefert setfenv keine Werte.
setmetatable(table,metatable)
Setzt die Metatabelle für die gegebene Tabelle. (Sie können die Metatabelle anderer Lua-Typen nicht ändern, nur von C aus.) Wenn metatable nil ist, wird die Metatabelle der gegebenen Tabelle entfernt. Wenn die originale Metatabelle ein "__metatable"-Feld hat, wird ein Fehler geworfen.
Diese Funktion liefert table.
tonumber(e[,base])

Versucht das Argument zu einer Zahl zu konvertieren. Wenn das Argument bereits eine zu einer Zahl konvertierbare Zahl oder Zeichenkette ist, liefert tonumber diese Zahl; andernfalls wird nil geliefert.
Ein optionales Argument gibt die anzunehmende Basis der Zahl an. Die Basis kann eine Ganzzahl von 2 bis 36 (inkl.) sein. Bei Basen über 10, steht der Buchstabe 'A' (sowohl in Groß- als auch Kleinschreibung) für 10, 'B' für 11 usf. bis 'Z' für 35. Bei der Basis 10 (Standard) können die Zahlen einen Dezimalteil und auch einen optionalen Exponentialteil haben (s. §?2.1).Bei anderen Basen sind lediglich natürliche Zahlen erlaubt.
tostring(e)
Nimmt ein Argument beliebigen Typs entgegen und konvertiert dieses in eine passende Zeichenkette. Für eine umfassende Kontrolle darüber, wie Zahlen konvertiert werden, verwenden Sie string.format.
Falls die Metatabelle von e ein "__tostring"-Feld besitzt, ruft tostring den entsprechenden Wert mit e als Argument auf und verwendet die Rückgabe des Aufrufs als dessen Rückgabe.
type(v)
Liefert den Wert des einzigen Arguments als Zeichenkette. Die möglichen Ergebnisse dieser Funktion sind "nil" (eine Zeichenkette, nicht den Wert nil), "number", "string", "boolean", "table", "function", "thread" oder "userdata".
unpack(list[,i[,j]])
Liefert die Elemente der gegebenen Tabelle. Diese Funktion ist äquivalent zu …
return list[i], list[i+1], ···, list[j]
… mit der Ausnahme, dass der o. g. Code nur für eine festgelegte Anzahl Elemente geschrieben werden kann. Standardmäßig ist i 1 und j ist die wie vom Längenoperator definierte Länge der Liste (s. §?2.5.5).
_VERSION
Eine globale Variable (keine Funktion), welche eine Zeichenkette mit der aktuellen Interpreter-Version enthält. Der aktuelle Inhalt dieser Variablen lautet "Lua 5.1".
xpcall(f,err)
Diese Funktion ist ähnlich wie pcall, mit dem Unterschied, dass Sie einen neuen Error-Handler setzen können.
xpcall ruft die Funktion f im geschützen Modus auf und verwendet err als Error-Handler. Kein Fehler innerhalb f wird geworfen; stattdessen fängt xpcall den Fehler, ruft die err-Funktion mit dem originalen Fehler-Objekt auf und liefert einen Statuscode. Der erste Rückgabewert ist der Statuscode (ein Wahrheitswert), welcher true ist, wenn der Aufruf fehlerfrei beendet wurde. In diesem Fall liefert xpcall auch alle Ergebnisse des Aufrufs nach dem ersten Rückgabewert. Im Fehlerfall liefert xpcall false und das Ergebnis von err.

5.2 Koroutinen-Verarbeitung

Die Koroutinen-bezogenen Operationen enthalten eine Unterbibliothek der Standardbibliothek und befinden sich innerhalb der coroutine-Tabelle. Siehe §?2.11für eine allgemeine Beschreibung zu Koroutinen.
coroutine.create(f)
Erzeugt eine neue Koroutine mit dem Inhalt f. f muss eine Lua-Function sein. Liefert diese neue Koroutine als Objekt vom Typ "thread" zurück.
coroutine.resume(co[,val1,···])
Startet die Ausführung der Koroutine co und setzt diese fort. Wenn Sie das erste Mal eine Koroutine wiederaufnehmen, wird deren Inhalt ausgeführt. Die Werte val1, ··· werden dem Inhalt der Funktion als Argumente übergeben. Wenn die Koroutine unterbrochen wurde, nimmt resume diese wieder auf; die Werte val1, ··· werden als Ergebnis der Unterbrechung durchgereicht.
Wenn die Koroutine ohne Fehler läuft, liefert resume true und alle Werte, welche yield übergeben wurden (sofern die Koroutine noch läuft) oder Werte, welche von der Funktion geliefert werden (wenn die Koroutine beendet wurde). Sollte ein Fehler auftreten liefert resume false und eine Fehlermeldung zurück.
coroutine.running()
Liefert die laufende Koroutine oder nil, wenn sie vom Haupt-Thread aufgerufen wird.

coroutine.status(co)
Liefert den Status der Koroutine co als Zeichenkette: "running", wenn die Koroutine läuft (dies ist der Fall, wenn status aufgerufen wurde); "suspended", wenn die Koroutine über einen Aufruf von yield angehalten wird, oder wenn sie noch nicht gestartet wurde; "normal" wenn die Koroutine aktiv ist, jedoch nicht läuft (dies ist der Fall, wenn eine andere Koroutine fortgesetzt wird) und "dead" wenn die Koroutine ihren Inhalt durchgeführt hat, oder mit einem Fehler beendet wurde.
coroutine.wrap(f)
Erstellt eine neue Koroutine mit dem Inhalt f. f muss eine Lua-Funktion sein. Liefert eine Funktion, welche die Koroutine bei jedem Aufruf wiederaufnimmt. Alle Argumente, welche der Funktion übergeben werden, fungieren als zusätzliche Argumente für resume. Liefert die gleichen Werte wie sie von resume zurückgegeben werden, mit Ausnahme des ersten Wahrheitswertes. Im Falle eines Fehlers wird dieser gemeldet.
coroutine.yield(···)
Unterbricht die Ausführung der aufrufenden Koroutine. Die Koroutine kann keine C-Funktionen, Metamethoden oder Iteratoren aufrufen.Alle Argumente, die yield übergeben werden, werden als separates Ergebnis an resume durchgereicht.

5.3 Module

Die Paketbibliothek bietet grundsätzliche Funktionen zum Laden und Erstellen von Modulen in Lua. Sie exportiert zwei ihrer Funktionen direkt in die globale Umgebung: require und module. Alles andere wird in eine Tabelle package exportiert.
module(name[,···])
Erstellt ein Modul. Falls sich eine Tabelle unter package.loaded[name] befindet, ist diese Tabelle das Modul. Wenn andernfalls eine globale Tabelle t mit dem gegebenen Namen existiert, ist diese Tabelle das Modul. Andernfalls wird eine neue Tabelle t erzeugt und als Wert des globalen name sowie package.loaded[name] gesetzt. Diese Funktion initialisiert auch t._NAME mit dem gegebenen Namen, t._M mit dem Modul (t selbst) und t._PACKAGE mit dem Paketnamen (der komplette Modulname ohne die letzte Komponente; s. u.). Schließlich setzt module t als neue Umgebung der aktuellen Funktion und den neuen Wert von package.loaded[name], so dass require t zurückliefert.

Falls name ein Kompositum ist (d. h. einer mit per Punkt getrennten Komponenten), erzeugt module Tabellen für jede Komponente (oder benutzt diese erneut, falls sie bereits existieren). Ist name beispielsweise a.b.c, dann speichert module die Tabelle im Feld c des Feldes b des globalen a.
Diese Funktion kann optionale options nach dem Modulnamen erhalten, wobei jede Option eine Funktion ist, welche auf das Modul angewendet wird.
require(modname)
Lädt das angegebene Modul. Die Funktion beginnt mit einer Suche in der package.loaded-Tabelle um herauszufinden, ob modname bereits geladen ist. Wenn das der Fall ist, liefert require den unter package.loaded[modname] gespeicherten Wert. Andernfalls versucht es einen Loader für das Modul zu finden.
Um einen Loader zu finden, wird require durch das package.loaders-Feld geführt. Durch das Verändern dieses Feldes können wir beeinflussen, wie require nach einem Modul sucht. Die folgende Erklärung basiert auf der Standardkonfiguration für package.loaders.
Zuerst fragt require package.preload[modname] ab. Wenn dies einen Wert hat, ist dieser Wert (welcher eine Funktion sein sollte) der Loader. Andernfalls sucht require nach einem Lua-Loader durch den in package.path gespeicherten Pad. Wenn das ebenfalls fehlschlägt, sucht es nach einem C-Loader mit Hilfe des unter package.cpath gespeicherten Pfades. Falls dies ebenfalls fehlschläft, versucht sie einen All-in-One-Loader (s. package.loaders).
Sobald ein Loader gefunden wurde, ruft require den Loader mit dem einzelnen Argument modname auf. Wenn der Loader irgendeinen Wert liefert, weist require den zurückgelieferten Wert package.loaded[modname] zu. Falls der Loader keinen Wert liefert und keinen Wert an package.loaded[modname] zugewiesen hat, weist require dem Eintrag true zu. In jedem Fall liefert require den finalen Wert von package.loaded[modname].
Falls ein Fehler beim Laden oder Ausführen des Moduls auftritt oder kein Loader für das

Modul gefunden werden kann, signalisiert require einen Fehler.
package.cpath
Der Pfad, welcher von require verwendet wird, um nach einem C-Loader zu suchen. Lua initialisiert den C-Pfad package.cpath auf die gleiche Weise wie den Lua-Pfad package.path unter Verwendung der Umgebungsvariable LUA_CPATH oder einen luaconf.h definierten Standardpfad.
package.loaded
Eine Tabelle, welche von require genutzt wird um zu kontrollieren, welche Module bereits geladen sind. Wenn Sie ein Modul modname einbinden und package.loaded[modname] nicht "false" ist, liefert require einfach den dort gespeicherten Wert.
package.loaders
Eine Tabelle, welche von require verwendet wird, um zu steuern, wie Module geladen werden.
Jeder Eintrag in dieser Tabelle ist eine Searcher-Funktion. Um nach einem Modul zu suchen ruft require jeden der Searcher in aufsteigender Reihenfolge mit dem Modulnamen (das Argument, welches an require übergeben wurde) als einzigem Parameter auf. Die Funktion kann eine weitere Funktion zurückliefert (das Modul loader) oder eine Zeichenkette, welche erklärt, wieso das Modul nicht gefunden werden konnte (oder nil, falls es nichts zu sagen gibt). Lua initialisiert diese Tabelle mit vier Funktionen.
Der erste Searcher sucht einfach nach einem Loader in der package.preload-Tabelle
Der zweite Searcher sucht nach einem Loader in Form einer Lua-Bibliothek unter Verwendung des unter package.path gespeicherten Pfades. Ein Pfad ist eine Folge von durch Semikolon getrennte Templates. Bei jedem Template ersetzt der Searcher jedes Fragezeichen im Template durch filename, was der Modulname ist, wobei jeder Punkt durch den Separator des Dateisystems (so etwas wie "/" unter Unix) ersetzt wurde; anschließend wird versucht, den sich ergebenden Dateinamen zu öffnen. Wenn der Lua-Pfad beispielsweise der Zeichenkette …
"./?.lua;./?.lc;/usr/local/?/init.lua"
… entspricht, versucht die Suche nach einer Lua-Datei des Moduls foo die Dateien ./foo.lua, ./foo.lc und /usr/local/foo/init.lua zu öffnen (in dieser Reihenfolge).
Der dritte Searcher sucht nach einem Loader in Form einer C-Bibliothek unter Verwendung des unter package.cpath gespeicherten Pfades. Wenn der C-Pfad beispielsweise der Zeichenkette …
"./?.so;./?.dll;/usr/local/?/init.so"
… entspricht, versucht der Searcher für das Modul foo die Dateien ./foo.so, ./foo.dll und /usr/local/foo/init.so zu öffnen (in dieser Reihenfolge). Sobald eine C-Bibliothek gefunden wird, benutzt der Searcher zuerst eine Funktion zum dynamischen Linken, um die Anwendung mit der Bibliothek zu linken. Dann wird versucht, eine C-Funktion innerhalb der Bibliothek zu finden, welche als Loader verwendet werden kann. Der Bezeichner dieser C-Funktion ist die Zeichenkette "luaopen_", verknüpft mit einer Kopie des Modulnamens, wobei jeder Punkt durch einen Unterstrich ersetzt wird. Darüber hinaus wird, wenn der Bezeichner einen Bindestrich enthält, dieser inkl. dem ersten vorangehenden Präfx entfernt. Wenn der Modulname beispielsweise a.v1-b.c ist, wird der Funktionsname luaopen_b_c sein.
Der vierte Searcher versucht einen All-in-One-Loader. Dieser durchsucht den C-Pfad nach einer Bibliothek für den ersten Bezeichner des gegebenen Moduls. Wenn beispielsweise das Modul a.b.c eingebunden wird, wird nach einer C-Bibliothek für a gesucht. Wird diese gefunden, wird nach einer öffnenden Funktion für das Untermodul gesucht; in unserem Beispiel wäre das luaopen_a_b_c. Durch diese Vorgehensweise kann ein Paket mehrere C-Untermodule in einer Bibliothek beinhalten, wobei jedes Untermodul seine originale öffnende Funktion behält.
package.loadlib(libname,funcname)
Linkt das Host-Programm dynamisch mit der C-Bibliothek libname. Innerhalb dieser Bibliothek wird nach einer funcname-Funktion gesucht und diese Funltion als C-Funktion zurückgeliefert. (funcname muss demnach die Spezifikation erfüllen (s. lua_CFunction)). Dies ist eine Low-Level-Funktion. Diese läuft völlig neben dem Paket- und Modulsystem. Im Gegensatz zu require führt diese keine Pfadsuche durch und fügt Erweiterungen nicht automatisch hinzu. libname musst der komplette Dateiname der C-Bibliothek sein, ggf.

inklusive Pfad und Dateierweiterung. funcname muss der exakte von der C-Bibliothek exportierte Name sein (was vom benutzten C-Compiler und Linker abhängen kann). Diese Funktion wird nicht von ANSI-C unterstützt. Insofern ist sie nur unter bestimmten Plattformen verfügbar (Windows, Linux, Mac OS X, Solaris, BSD und andere unixoide Systeme, welche den dlfcn-Standard unterstützen).
package.path
Der Pfad, welcher von require verwendet wird, um nach einem Lua-Loader zu suchen. Zu Beginn initialisiert Lua diese Variable mit dem Wert der Umgebungsvariable LUA_PATH oder mit einem in luaconf.h definierten Standardpfad, falls die Umgebungsvariable nicht definiert ist. Jedes ";;" im Wert der Umgebungsvariable wird durch den Standardpfad ersetzt.
package.preload
Eine Tabelle zum Speicher von Loadern für spezifische Module (s. require).
package.seeall(module)
Setzt eine Metatabelle für module, dessen __index-Feld auf die globale Umgebung zeigt, so dass dieses Modul Werte aus der globalen Umgebung erben kann. Zur Verwendung als Option für die Funktion module.

5.4 Zeichenkettenverarbeitung

Diese Bibliothek stellt generische Funktion zur Zeichenkettenverarbeitung zur Verfügung, unter anderem das Suchen und Extrahieren von Teil-Zeichenketten und die Mustersuche. Bei der Indizierung von Zeichenketten unter Lua befindet sich das erste Zeichen an Position 1 (nicht 0, wie unter C). Indizies können negativ sein und werden rückwärts interpretiert, also vom Ende der Zeichenkette an. Somit befindet sich das letzte Zeichen an Position -1 usw.
Die Zeichenketten-Bibliothek stellt alle Funktionen in der Tabelle string zur Verfügung. Sie setzt darüber hinaus eine Metatabelle für Zeichenketten, wobei das Feld __index auf die string-Tabelle zeigt. Dadurch können Sie die Zeichenketten-Funktionen in objektorientierem Stil verwenden. string.byte(s,i) kann beispielsweise als s:byte(i) geschrieben werden.
Die Zeichenketten-Bibliothek geht von einer Kodierung mit einem Byte pro Zeichen aus.
string.byte(s[,i[,j]])
Liefert den internen numerischen Code der Zeichen s[i], s[i+1], ···, s[j]. Der Standardwert für i ist 1; der Standardwert für j ist i.
Beachten Sie, dass numerische Codes nicht notwendigerweise über verschiedene Plattformen portabel sind.
string.char(···)
Erwartet 0 oder mehr Ganzzahlen. Liefert eine Zeichenkette mit einer Länge entsprechend der Anzahl der Argumente, wobei jedes Zeichen den gleichen numerischen Code besitzt, wie das korrespondierende Argument.
Beachten Sie, dass numerische Codes nicht notwendigerweise über verschiedene Plattformen portabel sind.
string.dump(function)
Liefert eine Zeichenkette, welche eine binäre Repräsentation der gegebenen Funktion enthält, so dass ein Späteres loadstring mit dieser Zeichenkette eine Kopie dieser Funktion liefert. function muss eine Lua-Funktion ohne gebundene Variablen sein.
string.find(s,pattern[,init[,plain]])
Sucht nach der ersten Übereinstimmung von pattern in der Zeichenkette s. Wenn eine Übereinstimmung gefunden wird, liefert find die Indizes von s, wo dieses Auftreten beginnt und endet; andernfalls liefert es nil. Ein drittes, optionales Argument init legt fest, wo mit der Suche begonnen werden soll; dessen Standardwert ist 1 und kann negativ sein. Der Wert true als viertes optioales Argument plain schaltet die Mustersuchs-Funktionalität ab, so dass die Funktion eine reine "finde Teil-Zeichenkette"-Operation durchführt, wobei keine Zeichen in pattern als "magisch" angesehen werden. Beachten Sie, dass wenn plain gegeben ist, init ebenfalls gegeben sein muss.
Wenn das Muster Treffer sichert, werden diese Werte bei einer erfolgreichen Übereinstimmung nach den zwei Indizes ebenfalls zurückgeliefert.
string.format(formatstring,···)
Liefert eine formatierte Version seiner variablen Anzahl an Argumenten nach der als

erstes Argument angegebenen Vorgabe (welche eine Zeichenkette sein muss). Die Formatierungs-Zeichenkette folgt den selben Regeln wie der printf-Familie der standardmäßigen C-Funktionen. Der einige Unterschied ist, dass die Optionen *, l, L, n, p und h nicht unterstützt werden und es eine zusätzliche Option q gibt. Die q-Option formatiert eine Zeichenkette in einer passenden Form, um sicher durch den Lua-Interpreter gelesen zu werden: Die Zeichenketten wird zwischen doppelten Anführungszeichen geschrieben und alle doppelten Anführungszeichen, Zeilenumbrüche, eingebetteten Nullen und Backslashes in der Zeichenkette werden beim Schreiben korrekt maskiert. Der Aufruf …

```
string.format('%q','a string with "quotes" and \n new line')
```

… wird beispielsweise folgende Zeichenkette erzeugen:

```
"a string with \"quotes\" and \
new line"
```

Die Optionen c, d, E, e, f, g, G, i, o, u, X und x erwarten alle eine Zahl als Argument, wohingegen q und s eine Zeichenkette erwarten.

Diese Funktion akzeptiert keine Zeichenketten, welche eingebettete Nullen enthalten, außer als Argument für die q-Option.

string.gmatch(s,pattern)

Liefert einen Iterator, welcher bei jedem Aufruf den nächsten Treffer aus pattern in der Zeichenkette s liefert. Falls pattern keine Treffer sichert, wird bei jedem Aufruf die komplette Übereinstimmung erzeugt.

Beispielsweise wird die folgende Schleife …

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

… über alle Worte der Zeichenkette s iterieren, wobei pro Zeile eines ausgegeben wird.

Das nächste Beispiel sammelt alle key=value-Paare der gegebenen Zeichenkette in einer Tabelle:

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s,"(%w+)=(%w+)") do
t[k] = v
end
```

Bei dieser Funktion fungiert ein '^' zu Beginn des Musters nicht als Anker, da dies die Iteration verhindern würde.

string.gsub(s,pattern,repl[,n])

Liefert eine Kopie von s, bei welcher alle (bzw. die ersten n, falls gegeben) Auftreten von pattern durch eine Ersetzungs-Zeichenkette, welche durch repl spezifiziert wurde und eine Zeichenkette, Tabelle oder Funktion sein kann, ersetzt wurden. gsub liefert als zweiten Wert auch die Gesamtzahl aufgetretener Treffer.

Wenn repl eine Zeichenkette ist, wird dessen Wert zur Ersetzung verwendet. Das Zeichen % dient der Maskierung: Jede Sequenz der Form %n in repl mit n zwischen 1 und 9 steht für den Wert der n-ten gesicherten Teil-Zeichenkette (s. u.). Die Sequenz %0 steht für die komplette Übereinstimmung. Die Sequenz %% steht für ein einzelnes %.

Wenn repl eine Tabelle ist, wird die Tabelle mit dem ersten Treffer als Schlüssel für jede Übereinstimmung abgefragt; falls das Muster keine Treffer angibt, wird die komplette Übereinstimmung als Schlüssel verwendet.

Wenn repl eine Funktion ist, wird diese für jede auftretende Übereinstimmung mit allen gesicherten Teil-Zeichenkette als Argumente in entsprechender Reihenfolge aufgerufen; falls das Muster keine Treffer angibt, wird die komplette Übereinstimmung als einziges Argument übergeben.

Wenn der von einer Tabellenabfrage oder Funktion zurückgelieferte Wert eine Zeichenkette oder Zahl ist, wird dies als Ersetzungs-Zeichenkette verwendet; andernfalls – wenn es false oder nil ist – findet keine Ersetzung statt (d. h. die Original-Übereinstimmung wird in der Zeichenkette belassen).

Hier sind ein paar Beispiele:

```
x = string.gsub("hello world","(%w+)","%1 %1")
```

```
-- x="hello hello world world"
x = string.gsub("hello world","%w+","%0 %0",1)
-- x="hello hello world"
x = string.gsub("hello world from Lua","(%w+)%s*(%w+)","%2 %1")
-- x="world hello Lua from"
x = string.gsub("home = $HOME, user = $USER","%$(%w+)",os.getenv)
-- x="home = /home/roberto, user = roberto"
x = string.gsub("4+5 = $return 4+5$","%$(.-)%$",function (s)
return loadstring(s)()
end)
-- x="4+5 = 9"
local t = {name="lua",version="5.1"}
x = string.gsub("$name-$version.tar.gz","%$(%w+)",t)
-- x="lua-5.1.tar.gz"
```

string.len(s)
Erhält eine Zeichenkette und liefert dessen Länge. Die leere Zeichenkette "" hat die Länge 0. Eingebettete Nullen werden gezählt, so dass "a\000bc\000" die Länge 5 hat.
string.lower(s)
Erhält eine Zeichenkette und liefert eine Kopie dieser Zeichenkette, wobei alle Großbuchstaben zu Kleinbuchstaben geändert werden. Alle anderen Zeichen bleiben ungeändert. Die Definition eines Großbuchstabens hängt von der aktuellen Sprache ab.
string.match(s,pattern[,init])
Sucht nach der ersten Übereinstimmung von pattern in der Zeichenkette s. Falls eine solche gefunden wird, liefert match die Treffer des Musters; andernfalls liefert es nil. Falls pattern keine Treffer spezifiziert, wird die komplette Übereinstimmung zurückgeliefert. Ein drittes, optionales numerisches Argument init gibt an, wo mit der Suche begonnen werden soll; dessen Standardwert ist 1 und kann negativ sein.
string.rep(s,n)
Liefert eine Zeichenkette, welche eine Verknüpfung von n Kopien der Zeichenkette s darstellt.
string.reverse(s)
Liefert eine Zeichenkette, welche die Zeichenkette s umgekehrt darstellt.
string.sub(s,i[,j])
Liefert eine Teil-Zeichenkette von s, welche sich von i bis j erstreckt; i und j können negativ sein. Falls j nicht angegeben wird, wird es als -1 angenommen (was das Gleiche wie die Länge der Zeichenkette ist). Im Speziellen liefert der Aufruf string.sub(s,1,j) ein Präfix von s der Länge j und string.sub(s,-i) liefert ein Suffix von s mit der Länge i.
string.upper(s)
Erhält eine Zeichenkette und liefert eine Kopie dieser Zeichenkette, wobei alle Kleinbuchstaben zu Großbuchstaben geändert werden. Alle anderen Zeichen bleiben ungeändert. Die Definition eines Kleinbuchstabens hängt von der aktuellen Sprache ab.

5.4.1 Muster

Eine Zeichenklasse repräsentiert eine Menge von Zeichen. Die folgenden Kombinationen sind zur Beschreibung einer Zeichenklasse erlaubt:

x: (soweit x keines der magischen Zeichen ^$()%.[]*+-?) ist) repräsentiert das Zeichen x selbst
.: (ein Punkt) repräsentiert alle Zeichen
%a: repräsentiert alle Buchstaben
%c: repräsentiert alle Steuerzeichen
%d: repräsentiert alle Ziffern
%l: repräsentiert alle Kleinbuchstaben
%p: repräsentiert alle Interpunktionszeichen
%s: repräsentiert alle Zwischenraumzeichen
%u: repräsentiert alle Großbuchstaben
%w: repräsentiert alle alphanumerischen Zeichen
%x: repräsentiert alle hexadezimalen Zeichen
%z: repräsentiert das Zeichen der Repräsentation 0

%x: (wobei x ein beliebiges, nicht-alphanumerisches Zeichen ist) Repräsentiert das Zeichen x. Dies ist das standardmäßige Vorgehen, um magische Zeichen zu maskieren. Allen Punktionszeichen (auch die nicht-magischen) kann ein '%' vorangestellt werden, um diese sich in einem Muster selbst repräsentieren zu lassen.
[set]: Repräsentiert die Klasse, welche eine Vereinigung aller Zeichen der Menge ist. Ein Bereich von Zeichen kann durch Trennung der Endzeichen mit einem '-' spezifiziert werden. Alle oben beschriebenen Klassen %x können ebenso als Komponenten der Menge benutzt werden. Alle anderen Zeichen der Menge repräsentieren sich selbst. Zum Beispiel repräsentieren [%w_] (oder [_%w]) alle alphanumerischen Zeichen inkl. dem Unterstrich, [0-7] repräsentiert oktale Ziffern und [0-7%l%-] repräsentiert die oktalen Ziffern inkl. Kleinbuchstaben und dem '-'-Zeichen.
Das Zusammenspiel von Bereichen und Klassen ist nicht definiert. Insofern haben Muster wie [%a-z] oder [a-%%] keine Bedeutung.
[^set]: Repräsentiert das Komplement von set, wobei set wie oben beschrieben interpretiert wird.

Für alle Klassen, die durch einen einzelnen Buchstaben repräsentiert werden (%a, %c etc.), steht der entsprechende Großbuchstabe für das Komplement der Klasse. Beispielsweise repräsentiert %S alle Nicht-Zwischenraumzeichen.
Die Definition von Buchstaben, Zwischenraum und anderen Zeichen hängt von der gegenwärtig gesetzten Sprache ab. Insbesondere kann die Klasse [a-z] möglicherweise nicht äquivalent zu %l sein.

Muster-Element
Ein Muster-Element kann sein …

… eine Klasse aus einzelnen Zeichen, welches auf jedes einzelne Zeichen der Klasse passt.
… eine Klasse aus einzelnen Zeichen, gefolgt von einem '*', was auf 0 oder mehr Wiederholungen der Zeichen aus dieser Klasse passt. Diese Wiederholungselemente passen immer auf die längstmögliche Sequenz.
… eine Klasse aus einzelnen Zeichen, gefolgt von einem '+', was auf 1 oder mehr Wiederholungen der Zeichen aus dieser Klasse passt. Diese Wiederholungselemente passen immer auf die längstmögliche Sequenz.
… eine Klasse aus einzelnen Zeichen, gefolgt von einem '-', was ebenfalls auf 0 oder mehr Wiederholungen der Zeichen aus dieser Klasse passt. Im Gegensatz zu '*', passen diese Wiederholungselemente immer auf die kürzestmögliche Sequenz.
… eine Klasse aus einzelnen Zeichen, gefolgt von einem '?', was auf 0 oder 1 Auftreten eines Zeichens dieser Klasse passt.
… %n für n von 1 bis 9; dies passt auf eine Teil-Zeichenkette, welche identisch zur n-ten gefundenen Zeichenkette ist (s. darunter)
… %bxy, wobei x und y zwei verschiedene Zeichen sind; dies passt auf Zeichenketten, welche mit x beginnen und mit y enden und x und y balanciert sind. Das bedeutet, dass wenn man die Zeichenkette von links nach rechts liest, für ein x +1 zählt und für ein y -1, so ist das letzte y das erste y für das der Zähler 0 erreicht. Beispielsweise passt %b() auf Ausdrücke mit balancierten Klammern.

Muster
Ein Muster ist eine Sequenz von Muster-Elementen. Ein '^' am Beginn eines Musters verankert die Übereinstimmung am Beginn der zu untersuchenden Zeichenkette. Ein '$' am Ende eines Musters verankert die Übereinstimmung am Ende der zu untersuchenden Zeichenkette. An anderen Stellen haben '^' und '$' keine spezielle Bedeutung und stehen für sich selbst.

Treffer
Ein Muster kann in Klammern eingeschlossene Teilmuster beinhalten; diese beschreiben Treffer. Wenn eine Übereinstimmung gefunden wird, wird die Teil-Zeichenkette der Zeichenkette, welche einen Treffer erzeugt, für zukünftige Nutzung gespeichert. Treffer

werden entsprechend ihrer linken Klammer nummeriert. Im Muster "(a*(.)%w(%s*))" wird beispielsweise der Teil der Zeichenkette, welcher auf "a*(.)%w(%s*)" passt, als erster Treffer gespeichert (und hat somit die Nummer 1); die Zeichen, welche auf "." passen, werden als Nummer 2 abgelegt und der Teil, welcher auf "%s*" passt, hat die Nummer 3.
() ermittelt als Spezialfall die aktuelle Zeichenketten-Position (eine Nummer). Wenn wir beispielsweise das Muster "()aa()" auf die Zeichenkette "flaaap" anwenden, bekommen wir die zwei Treffer 3 und 5.
Ein Muster kann keine Nullen enthalten. Benutzen Sie stattdessen %z.

5.5 Tabellenverarbeitung

Diese Bibliothek bietet generische Funktionen zur Tabellenverarbeitung. Sie stellt alle ihre Funktionen innerhalb der Tabelle table zur Verfügung..
Die meisten Funktionen der Tabellenbibliothek gehen davon aus, dass die Tabelle ein Feld oder eine Liste repräsentiert. Für diese Funktionen gilt, dass wenn wir über die "Länge" einer Tabelle sprechen, wir das Ergebnis des Längenoperators meinen.
table.concat(table[,sep[,i[,j]]])
Bei Übergabe eines Feldes, dessen Elemente alle Zeichenketten oder Zahlen sind, wird table[i]..sep..table[i+1] ··· sep..table[j] zurückgegeben. Der Standardwert für sep ist die leere Zeichenkette, der Standard für i ist 1 und der Standard für j ist die Länge der Tabelle. Falls i größer als j ist, wird die leere Zeichenkette zurückgeliefert.
table.insert(table,[pos,]value)
Fügt table das Element value an Position pos hinzu, wobei andere Elemente falls notwendig nach oben verschoben werden, um Platz zu schaffen. Der Standardwert für pos ist n+1, wobei n die Länge der Tabelle ist (s. §?2.5.5),so dass der Aufruf table.insert(t,x) x an das Ende der Tabelle t hinzufügt.
table.maxn(table)
Liefert den größten positiven numerischen Index der gegebenen Tabelle oder 0, falls die Tabelle keine positiven numerischen Indizes besitzt. (Um dies zu erledigen, führt die Funktion eine lineare Traversierung der gesamten Tabelle durch.)
table.remove(table[,pos])
Entfernt das Element an der Position pos aus table, wobei andere Elemente falls notwendig nach unten verschoben werden, um Platz zu schaffen. Liefert den Wert des entfernten Elements. Der Standardwert für pos ist n, wobei n die Länge der Tabelle ist, so dass der Aufruf table.remove(t) das letzte Element der Tabelle t entfernt.
table.sort(table[,comp])
Sortiert die Elemente der Tabelle in-place mit der gegebenen Sortierung von table[1] nach table[n], wobei n die Länge der Tabelle ist. Wenn comp gegeben ist, muss dies eine Funktion sein, welche zwei Tabellenelemente annehmen kann und true liefert, wenn das erste Element kleiner als das zweite ist (so dass not comp(a[i+1],a[i]) nach der Sortierung true wird). Wenn comp nicht angegeben ist, wird stattdessen der Standardoperator < von Lua benutzt.
Das Sortierverfahren ist nicht stabil; d. h., dass wenn Elemente von der gegebenen Sortierung als gleich betrachtet werden, deren relative Position zueinander von der Sortierung verändert werden kann.

5.6 Mathematische Funktionen

Diese Bibliothek ist ein Wrapper zur math-Standardbibliothek von C. Dieser stellt alle seine Funktionen innerhalb der Tabelle math zur Verfügung.
math.abs(x)
Liefert den absoluten Betrag von x.
math.acos(x)
Liefert den Arkuskosinus von x (in Radiant).
math.asin(x)
Liefert den Arkussinus von x (in Radiant).
math.atan(x)
Liefert den Arkustangens von x (in Radiant).
math.atan2(y,x)
Liefert den Arkustangens von y/x (in Radiant), aber benutzt die Vorzeichen beider

Parameter, um den korrekten Quadranten des Ergebnisses zu ermitteln. (Der Fall, dass x gleich 0 ist, wird ebenfalls korrekt behandelt.)
math.ceil(x)
Liefert die kleinste Ganzzahl größer oder gleich x.
math.cos(x)
Liefert den Kosinus von x (in Radiant).
math.cosh(x)
Liefert den Kosinus Hyperbolicus von x (in Radiant).
math.deg(x)
Liefert den Winkel x (in Radiant) in Grad.
math.exp(x)
Liefert den Wert ex.
math.floor(x)
Liefert die größte Ganzzahl kleiner oder gleich x.
math.fmod(x,y)
Liefert den Rest der Division x durch y, welche den Quotienten auf 0 rundet.
math.frexp(x)
Liefert m und e, welche x = m2e lösen; e ist eine Ganzzahl und der absolute Betrag von m ist im Intervall [0.5,1) (oder 0, wenn x gleich 0 ist).
math.huge
Der Wert HUGE_VAL, welcher größer oder gleich jedem anderen Zahlenwert ist.
math.ldexp(m,e)
Liefert m2e (e sollte eine Ganzzahl sein).
math.log(x)
Liefert den natürlichen Logarithmus von x.
math.log10(x)
Liefert den dekadischen Logarithmus von x.
math.max(x,···)
Liefert den größten Wert seiner Argumente.
math.min(x,···)
Liefert den kleinsten Wert seiner Argumente.
math.modf(x)
Liefert zwei Zahlen: Den ganzzahligen und den gebrochenen Anteil von x.
math.pi
Der Wert von Pi.
math.pow(x,y)
Liefert xy. (Sie können diesen Wert auch durch den Ausdruck x^y berechnen.)
math.rad(x)
Liefert den Winkel x (in Grad) in Radiant.
math.random([m[,n]])
Diese Funktion ist ein Wrapper zum einfachen Pseudozufallszahlen-Generator rand, welcher von ANSI C zur Verfügung gestellt wird. (Es gibt keine Gewährleistung für dessen statistische Eigenschaften.)
Wird diese Funktion ohne Argumente aufgerufen, liefert sie eine gleichverteilte, reelle Pseudozufallszahl aus dem Intervall [0,1). Wird diese Funktion mit einer Ganzzahl m aufgerufen, liefert math.random eine gleichverteilte, ganzzahlige Pseudozufallszahl aus dem Intervall [1,m]. Wird diese Funktion mit zwei Ganzzahlen m und n aufgerufen, liefert math.random eine gleichverteilte, ganzzahlige Pseudozufallszahl aus dem Intervall [m,n].

math.randomseed(x)
Setzt x als "Seed" für den Pseudozufallszahlen-Generator: Gleiche Seeds produzieren gleiche Zahlensequenzen.
math.sin(x)
Liefert den Sinus von x (in Radiant).
math.sinh(x)
Liefert den Sinus Hyperbolicus von x (in Radiant).
math.sqrt(x)
Liefert die Quadratwurzel von x. (Sie können diesen Wert auch durch den Ausdruck

x^0.5 berechnen.)
math.tan(x)
Liefert den Tangens von x (in Radiant).
math.tanh(x)
Liefert den Tangens Hyperbolicus von x (in Radiant).

5.7 Ein- und Ausgabe

Die I/O-Bibliothek stellt zwei verschiedene Arten der Dateiverarbeitung zur Verfügung. Die erste nutzt implizite Dateideskriptoren; d. h. es gibt Operationen, um eine standardmäßige Ein- und Ausgabedatei zu setzen und alle Ein-/Ausgabeoperationen beziehen sich auf diese Standard-Dateien. Die zweite Art nutzt explizite Dateideskriptoren.
Bei der Verwendung impliziter Dateideskriptoren werden alle Operationen durch die Tabelle io angeboten. Bei der Verwendung expliziter Dateideskriptoren liefert die Operation io.open einen Dateideskriptor und alle Operationen werden als dessen Methoden angeboten.
Die Tabelle io stellt außerdem drei vordefinierte Dateideskriptoren mit deren gewöhnlichen Bedeutung aus C zur Verfügung: io.stdin, io.stdout und io.stderr. Die I/O-Bibliothek schließt diese Dateien niemals.
Sofern nicht anders angegeben, liefert alle I/O-Funktionen im Falle eines Fehlers nil (und eine Fehlernachricht als zweites Ergebnis und einen systemabhängigen Fehlercode als drittes Ergebnis) und einen von nil verschiedenen Wert im Erfolgsfall.
io.close([file])
Äquivalent zu file:close(). Ohne file wird die Standard-Ausgabedatei geschlossen.
io.flush()
Äquivalent zu file:flush für die Standard-Ausgabedatei.
io.input([file])
Wenn diese Funktion mit einem Dateinamen aufgerufen wird, öffnet sie die angegebene Datei (im Textmodus) und setzt dessen Handle als Standard-Eingabedatei. Wenn diese Funktion mit einem Datei-Handle aufgerufen wird, setzt sie dieses als Standard-Eingabedatei. Bei einem Aufruf ohne Parameter liefert sie die gegenwärtige Standard-Eingabedatei.
Im Falle eines Fehlers wirft diese Funktion diesen, anstatt einen Fehlercode zu liefern.
io.lines([filename])
Öffnet den angegebenen Dateinamen im Lesemodus und liefert einen Iterator, welcher bei jedem Aufruf eine neue Zeile der Datei liefert. Insofern wird eine Konstruktion wie … for line in io.lines(filename) do body end
… über alle Zeilen der Datei iterieren. Sobald der Iterator das Ende der Datei erkennt, liefert er nil (um die Schleife zu beenden) und schließt die Datei automatisch.
Der Aufruf io.lines() (ohne Dateiname) ist äquivalent zu io.input():lines(); d. h. er iteriert über die Zeilen der Standard-Eingabedatei. In diesem Fall wird die Datei nach Beendigung der Schleife nicht geschlossen.
io.open(filename[,mode])
Diese Funktion öffnet eine Datei im mit der Zeichenkette mode angegebenen Modus. Sie liefert ein neues Datei-Handle, oder – im Falle eines Fehlers – nil und eine Fehlernachricht.
Die mode-Zeichenkette kann eine der folgenden sein:

"r": lesen (Standard)
"w": schreiben
"a": anhängen
"r+": aktualisieren; alle vorherigen Daten bleiben erhalten
"w+": aktualisieren; alle vorherigen Daten gehen verloren
"a+": anhängen + aktualisieren; vorherige Daten bleiben erhalten und Schreiben ist nur am Ende der Datei möglich

Die mode-Zeichenkette kann auch ein 'b' am Ende haben, was unter manchen Systemen benötigt wird, um die Datei im Binärmodus zu öffnen. Diese Zeichenkette ist exakt die selbe, welche in der standardmäßigen C-Funktion fopen verwendet wird.

io.output([file])
Ähnlich wie io.input, arbeitet jedoch auf der Standard-Ausgabedatei.
io.popen(prog[,mode])
Öffnet das Programm prog in einem separaten Prozess und liefert ein Datei-Handle, welches Sie verwenden können, um Daten von diesem Programm zu lesen (wenn mode "r" ist; Standard) oder um Daten zu schreiben (wenn mode "w" ist).
Diese Funktion ist systemabhängig und nicht unter allen Plattformen verfügbar.
io.read(···)
Äquivalent zu io.input():read.
io.tmpfile()
Liefert ein Handle für eine temporäre Datei. Die Datei wird im Aktualisierungsmodus geöffnet und automatisch entfernt, wenn das Programm endet.
io.type(obj)
Prüft, ob obj ein gültiges Datei-Handle ist. Liefert die Zeichenkette "file", wenn obj ein offenes Datei-Handle ist, "closed file", wenn obj ein geschlossenes Datei-Handle ist, oder nil, wenn obj kein Datei-Handle ist.
io.write(···)
Äquivalent zu io.output():write.
file:close()
Schließt file. Beachen Sie, dass Dateien automatisch geschlossen werden, wenn deren Handles von der automatischen Speicherbereinigung gesammelt werden, was jedoch eine unbestimmte Zeit dauern kann.
file:flush()
Speichert jegliche geschriebenen Datei für file.
file:lines()
Liefert einen Iterator, welcher bei jedem Aufruf eine neue Zeile der Datei liefert. Somit wird eine Konstruktion wie …
for line in file:lines() do body end
… über alle Zeilen der Datei iterieren. (Im Gegensatz zu io.lines schließt diese Funktion die Datei nach Beendigung der Schleife nicht.
file:read(···)
Liest die Datei file im angegebenen Format, welches spezifiziert, was gelesen werden soll. Diese Funktion liefert für jedes Format eine Zeichenkette (oder Nummer) mit den gelesenen Zeichen oder nil, wenn keine Daten im angegebenen Format gelesen werden konnten. Bei einem Aufruf ohne Formate wird ein Standardformat verwendet, welches die komplette nächste Zeile liest (s. u.).
Die verfügbaren Formate lauten wie folgt:

"*n": Nummer lesen; dies ist das einzige Format, welches eine Zahl anstatt einer Zeichenkette liefert
"*a": Liest die komplette Datei, beginnend bei der aktuellen Position. Am Ende jeder Datei liefert es die leere Zeichenkette.
"*l": Liest die nächste Zeile (unter Überspringen des Zeilenendes) und liefert nil am Ende der Datei. Dies ist das Standardformat.
Nummer: Liest eine Zeichenkette bis zu dieser Anzahl von Zeichen und liefert nil am Ende der Datei. Wenn die Zahl 0 ist, wird nichts gelesen und die leere Zeichenkette zurückgeliefert, oder nil am Ende der Datei.

file:seek([whence][,offset])
Setzt und liefert die Position des Dateizeigers, gemessen vom Anfang der Datei bis zur durch offset gegebenen Position und einer durch die Zeichenkette whence spezifizierten Basis:

"set": Basis ist Position 0 (Anfang der Datei)
"cur": Basis ist aktuelle Position
"end": Basis ist Ende der Datei

Im Erfolgsfall liefert die Funktion seek die finale Position des Dateizeigers, gemessen in

Byte vom Beginn der Datei. Falls diese Funktion fehlschlägt, liefert sie nil und eine den Fehler beschreibende Zeichenkette.
Der Standardwert für whence ist "cur" und für offset 0. Somit liefert der Aufruf file:seek() die aktuelle Position des Dateizeigers ohne diese zu ändern; der Aufruf file:seek("set") setzt die Position auf den Anfang der Datei (und liefert 0); der Aufruf file:seek("end") setzt schließlich die Position an das Ende der Datei und liefert deren Größe.
file:setvbuf(mode[,size])
Setzt den Puffermodus für eine Ausgabedatei. Es sind drei Modi verfügbar:

"no": keine Pufferung; das Ergebnis jeder Ausgabe-Operation erscheint sofort
"full": volle Pufferung; Ausgabeoperationen werden nur durchgeführt, wenn der Puffer voll ist (oder wenn Sie explizit flush auf die Datei anwenden (s. io.flush))
"line": Zeilenpufferung; Ausgabe wird gepuffert, bis ein Zeilenumbruch oder eine Eingabe aus einer speziellen Datei (z. B. ein Terminal) erfolgt

In den letzten beiden Fällen spezifiziert size die Puffergröße (in Byte). Der Standard ist eine angemessene Größe.
file:write(···)
Schreibt den Wertes jedes Arguments in file. Die Argumente müssen Zeichenketten oder Zahlen sein. Um andere Werte zu schreiben, verwenden Sie tostring oder string.format vor write.

5.8 Betriebssystem

Diese Bibliothek ist über die os-Tabelle implementiert.
os.clock()
Liefert eine Annäherung der CPU-Zeit in Sekunden, welche durch das Programm benutzt wird.
os.date([format[,time]])
Liefert eine Zeichenkette oder Tabelle, welche das Datum und die Uhrzeit entsprechend der angegebenen Zeichenkette format beinhalten.
Wenn das Argument time angegeben ist, wird diese Zeit formatiert (s. os.time-Funktion für eine Beschreibung dieses Wertes). Andernfalls formatiert date die aktuelle Zeit.
Wenn format mit '!' beginnt, wird das Datum in koordinierter Weltzeit formatiert. Wenn format nach diesem optionalen Zeichen die Zeichenkette "*t" ist, liefert date eine Tabelle mit folgenden Feldern: year (vierstellig), month (1-12), day (1-31), hour (0-23), min (0-59), sec (0-61), wday (Wochentag; Sonntag ist 1), yday (Tag des Jahres) und isdst (Sommerzeit, ein bool'scher Wert).
Falls format nicht "*t" lautet, liefert date das Datum als Zeichenkette mit einer Formatierung nach den gleichen Regeln wie die der C-Funktion strftime.
When called without arguments, date returns a reasonable date and time representation that depends on the host system and on the current locale (d. h. os.date() ist äquivalent zu os.date("%c")).
os.difftime(t2,t1)
Liefert die Anzahl Sekunden von t1 zu t2. Unter POSIX, Windows und einigen anderen Systeme entspricht dieser Wert exakt t2-t1.
os.execute([command])
Diese Funktion ist äquivalent zur C-Funktion system. Sie übergibt command zur Ausführung über eine Betriebssystem-Shell. Sie liefert einen Statuscode, welcher systemabhängig ist. Falls command nicht angegeben wird, liefert sie nicht-0 wenn eine Shell verfügbar ist und andernfalls 0.
os.exit([code])
Ruft die C-Funktion exit mit einem optionalen code zur Beendigung des Hostprogramms auf. Der Standardwert für code ist der "success"-Code.
os.getenv(varname)
Liefert den Wert der Prozess-Umgebungsvariable varname, oder nil falls die Variable nicht definiert ist.
os.remove(filename)
Löscht die Datei oder das Verzeichnis mit dem angegebenen Namen. Ordner müssen leer sein, um entfernt werden zu können. Falls diese Funktion fehlschlägt, liefert diese nil und

eine den Fehler beschreibende Zeichenkette.
os.rename(oldname,newname)
Benennt die Datei oder das Verzeichnis mit dem Namen oldname zu newname um. Falls diese Funktion fehlschlägt, liefert diese nil und eine den Fehler beschreibende Zeichenkette.
os.setlocale(locale[,category])
Setzt die aktuelle Sprache des Programms. locale ist eine Zeichenkette, welche die Sprache spezifiziert; category ist eine optionale Zeichenkette, welche beschreibt, welche Kategorie geändert werden soll: "all", "collate", "ctype", "monetary", "numeric" oder "time"; die Standardkategorie ist "all". Die Funktion liefert den Bezeichner der neuen Sprache oder nil, falls die Anfrage nicht verarbeitet werden kann.
Wenn locale die leere Zeichenkette ist, wird die aktuelle Sprache zur per Implementierung definierten nativen Sprache gesetzt. Falls locale die Zeichenkette "C" ist, wird die aktuelle Sprache auf die standardmäßige C-Spracheinstellung gesetzt.
Wenn die Funktion mit nil als erstem Argument aufgerufen wird, liefert sie lediglich die aktuelle Spracheinstellung für die gegebene Kategorie zurück.
os.time([table])
Liefert bei Aufruf ohne Argumente die aktuelle Zeit oder die Zeit, welche durch die übergebene Tabelle spezifiziert wird. Diese Tabelle muss die Felder year, month und day besitzen und kann (zusätzlich) die Felder hour, min, sec und isdst haben (s. os.date-Funktion für eine Beschreibung dieser Felder).
Die Bedeutung des zurückgelieferten Werts hängt von Ihrem System ab. Unter POSIX, Windows und einigen anderen System zählt diese Nummer die Anzahl der Sekunden seit einer bestimmten Startzeit (die "Epoche"). Unter anderen System ist die Bedeutung nicht definiert und die von time gelieferte Nummer kann ausschließlich als Argument für date und difftime dienen.
os.tmpname()
Liefert eine Zeichenkette mit einem Dateinamen, welcher für eine temporäre Datei genutzt werden kann. Die Datei muss vor deren Benutzung explizit geöffnet und explizit geschlossen werden, wenn sie nicht länger benötigt wird.
Unter manchen Systemen (POSIX) erzeugt diese Funktion auch die Datei mit diesem Namen, um Sicherheitsrisiken zu verhindern. (Jemand anderes könnte die Datei zwischen dem Erhalten des Namens und der Erzeugung der Datei mit falschen Zugriffsrechten erzeugen.) Sie müssen nach wie vor die Datei öffnen, um sie zu nutzen und schließen (auch, wenn Sie sie nicht nutzen).
Sofern möglich, sollten Sie io.tmpfile bevorzugen, welche die Datei automatisch entfernt, wenn das Programm endet.

5.9 Die Debug-Bibliothek

Diese Bibliothek stellt die Funktionalität der Debug-Schnittstelle Lua-Programmen zur Verfügung. Sie sollten vorsichtig sein, wenn Sie diese Bibliothek benutzen. Die hier angebotenen Funktionen sollten ausschließlich zum Debuggen und ähnlichen Zwecken wie dem Profiling verwendet werden. Bitte widerstehen Sie der Versuchung, diese als gewöhnliche Programmtools zu verwenden: Diese können sehr langsam sein. Darüber hinaus verletzen einige dieser Funktionen ein paar Annahmen über Lua-Code (z. B. dass lokale Variablen einer Funktion nicht von außerhalb erreichbar sind, oder dass Metatabellen nicht durch Lua-Code geändert werden können) und können insofern an und für sich sicheren Code kompromittieren.
Alle Funktionen dieser Bibliothek werden innerhalb der debug-Tabelle zur Verfügung gestellt. Alle Funktionen, welche auf Threads operieren, haben ein optionales erstes Argument, welches der zu bearbeitende Thread ist. Standard ist immer der aktuelle Thread.
debug.debug()
Startet einen interaktiven Modus mit dem Benutzer, wobei jede von diesem eingegebene Zeichenkette ausgeführt wird. Durch die Verwendung einfacher Befehle und anderer Debug-Funktionen kann der Benutzer globale und lokale Variablen inspizieren, deren Werte ändern, Ausdrücke auswerten usw. Eine Zeile, welche lediglich den Begriff cont enthält, beendet diese Funktion, so dass der Aufrufer dessen Ausführung fortsetzt.
Beachen Sie, dass die Befehle für debug.debug sich in keinem lexikalischen

Gültigkeitsbereich irgendwelcher Funktionen befinden, so dass diese keinen direkten Zugriff auf lokale Variablen haben.
debug.getfenv(o)
Liefert die Umgebung des Objekts o.
debug.gethook([thread])
Liefert die aktuellen Hook-Einstellungen des Threads in drei Werten: Die aktuelle Hook-Funktion, die aktuelle Hook-Maske und die aktuelle Anzahl der Hooks (wie von der debug.sethook-Funktion gesetzt).
debug.getinfo([thread,]function[,what])
Liefert eine Tabelle mit Informationen zu einer Funktion. Sie können direkt die Funktion angeben, oder aber eine Nummer als Wert von function, welche die laufende Funktion der Ebene function des Aufrufstapels des gegebenen Threads bezeichnet: Level 0 ist die aktuelle Funktion (getinfo selbst); Level 1 ist die Funktion, welche getinfo aufgerufen hat usw. Falls function eine größere Zahl als die der aktiven Funktionen ist, liefert getinfo nil.

Die gelieferte Tabelle enthält alle von lua_getinfo zurückgegebenen Felder mit der Zeichenkette what, welche beschreibt, welche Felder zu füllen sind. Das standardmäßige Verhalten von what ist es, alle verfügbaren Informationen außer die Tabelle der gültigen Zeilen zu liefern. Wenn angegeben, fügt die Option 'f' ein Feld mit dem Namen func mit der Funktion selbst hinzu. Wenn angegeben, fügt die Option 'L' ein Feld mit dem Namen activelines mit der Tabelle gültiger Zeilen hinzu.
Der Ausdruck debug.getinfo(1,"n").name liefert beispielsweise eine Tabelle mit einem Bezeichner für die aktuelle Funktion, sofern ein passender Name gefunden werden kann und der Ausdruck debug.getinfo(print) liefert eine Tabelle mit allen verfügbaren Informationen zur print-Funktion.
debug.getlocal([thread,]level,local)
Diese Funktion liefert den Namen und Wert der lokalen Variable mit dem Index local der Funktion auf Ebene level des Aufrufstapels. (Der erste Parameter oder die erste lokale Variable hat Index 1 usw., bis zur letzten aktiven lokalen Variablen.) Die Funktion liefert nil, falls es keine lokale Variable mit dem gegebenen Index gibt und wirft einen Fehler, wenn sie mit einem level außerhalb des zulässigen Wertebereichs aufgerufen wird. (Sie können getinfo aufrufen, um zu prüfen, ob eine Ebene gültig ist.)
Variablennamen, welche mit '(' (geöffneten Klammern) beginnen, repräsentieren interne Variablen (Schleifen-Zählvariablen, temporäre Variablen und lokale Variablen aus C-Funktionen).
debug.getmetatable(object)
Liefert die Metatabelle des gegebenen object oder nil, falls es keine Metatabelle besitzt.
debug.getregistry()
Liefert die Registry-Tabelle (s. §3.5).
debug.getupvalue(func,up)
Diese Funktion liefert den Namen und Wert der gebundenen Variable mit dem Index up der Funktion func. Die Funktion liefert nil, falls es keine gebundene Variable mit dem gegebenen Index gibt.
debug.setfenv(object,table)
Setzt die Umgebung des gegebenen object auf die gegebene table. Liefert object.
debug.sethook([thread,]hook,mask[,count])
Setzt die gegebene Funktion als Hook. Die Zeichenkette mask und die Zahl count beschreiben, wann der Hook aufgerufen wird. Die Zeichenkette mask kann folgende Zeichen mit entsprechender Bedeutung beinhalten:

"c": Der Hook wird jedes Mal aufgerufen, wenn Lua eine Funktion aufruft.
"r": Der Hook wird jedes Mal aufgerufen, wenn Lua aus einer Funktion zurückkehrt.
"l": Der Hook wird jedes Mal aufgerufen, wenn Lua eine neue Zeile Code verarbeitet.

Mit einem von 0 verschiedenen count wird der Hook nach jeder count-ten Anweisung aufgerufen.
Bei einem Aufruf ohne Argumente schaltet debug.sethook den Hook aus.
Wenn der Haltepunkt aufgerufen wird, ist dessen erster Parameter eine Zeichenkette,

welche das getriggerte Ereignis beschreibt: "call", "return" (oder "tail return", wenn eine Rückgabe aus einer Endrekursion simuliert wird), "line" und "count". Für Zeilen-Ereignisse erhält der Hook darüber hinaus die neue Zeilennummer als zweiten Parameter. Innerhalb eines Hooks können Sie getinfo mit Level 2 aufrufen, um weitere Informationen der laufenden Funktion zu erhalten (Level 0 ist die getinfo-Funktion und Level 1 ist die Hook-Funktion), außer das Ereignis ist "tail return". In diesem Fall simuliert Lua die Rückgabe lediglich und ein Aufruf von getinfo wird ungültige Daten liefern.
debug.setlocal([thread,] level,local,value)
Diese Funktion weist den Wert value der Variablen local mit dem Index local der Funktion der Ebene level des Stapelspeichers zu. Die Funktion liefert nil, falls es keine lokale Variable mit dem gegebenen Index gibt und wirft einen Fehler, wenn sie mit einem level außerhalb des zulässigen Wertebereichs aufgerufen wird. (Sie können getinfo aufrufen, um zu prüfen, ob eine Ebene gültig ist.) Andernfalls liefert sie den Namen der lokalen Variablen.
debug.setmetatable(object,table)
Setzt die Metatabelle des gegebenen object auf table (welche nil sein kann).
debug.setupvalue(func,up,value)
Diese Funktion weist den Wert value der gebundenen Variable mit dem Index up der Funktion func zu. Die Funktion liefert nil, falls es keine gebundene Variable mit dem gegebenen Index gibt. Andernfalls liefert sie den Namen der gebundenen Variable.
debug.traceback([thread,] [message[, level]])
Liefert eine Zeichenkette mit einem Stacktrace des Aufrufstapels. Eine optionale message-Zeichenkette wird dem Stacktrace vorangestellt. Eine optionale level-Nummer legt fest, auf welcher Ebene die Ablaufverfolgung begonnen werden soll (Standard ist 1, die aufrufende Funktion von traceback).

6 Standalone-Lua

Obwohl Lua als Skriptsprache entworfen wurde, um eingebettet in C-Programmen zu laufen, wird es ebenfalls häufig als eigenständige Sprache verwendet. Ein Interpreter für Lua als eigenständige Sprache – lua genannt – wird mit der Standarddistribution angeboten. Der Interpreter beinhaltet alle Standardbibliotheken inklusive der Debug-Bibliothek. Benutzung:
lua [Optionen] [Skript [Argumente]]
Die Optionen sind:

-e stat: Führt die Zeichenkette stat aus.
-l mod: mod einbinden
-i: Startet den interaktiven Modus nach der Skriptausführung.
-v: Gibt Versions-Informationen aus.
--: Beendet das Behandeln von Optionen.
-: Führt stdin als Datei aus und beendet das Behandeln von Optionen.

Nach der Verarbeitung der Optionen führt lua das angegebene Skript aus und übergibt diesem die angegebenen Argumente als Zeichenketten. Wenn der Aufruf ohne Argumente erfolgt, verhält sich lua wie lua -v -i, sofern die Standardeingabe (stdin) eine Konsole ist und andernfalls wie lua -.
Bevor irgendein Argument verarbeitet wird, prüft der Interpreter auf eine Umgebungsvariable LUA_INIT. Wenn deren Format @filename ist, dann führt lua die Datei aus. Andernfalls führt lua die Zeichenkette selbst aus.
Alle Optionen werden in ihrer entsprechenden Reihenfolge verarbeitet, außer -i. Zum Beispiel wird der Aufruf …
$ lua -e'a=1' -e 'print(a)' script.lua
… zuerst a auf 1 setzen, dann den Wert von a ausgeben (welcher '1' ist) und schließlich die Datei script.lua ohne Argumente ausführen. (Hier entspricht $ der Eingabeaufforderung. Ihre kann anders aussehen.)
Vor der Ausführung des Skripts sammelt lua alle Argumente der Kommandozeile in einer globalen Tabelle namens arg. Der Skriptname ist am Index 0 gespeichert, das erste

Argument nach dem Skriptnamen unter dem Index 1 usw. Alle Argumente vor dem Skriptnamen (dieser entspricht dem Interpreter-Namen plus den Optionen) befinden sich unter negativen Indizes. Zum Beispiel wird der Interpreter bei …
$ lua -la b.lua t1 t2
… zuerst die Datei a.lua ausführen, dann eine Tabelle erstellen …
arg = { [-2] = "lua", [-1] = "-la",
[0] = "b.lua",
[1] = "t1", [2] = "t2" }
… und schließlich die Datei b.lua ausführen. Das Skript wird mit den Argumenten arg[1], arg[2], ··· aufgerufen; es kann ebenso mit dem "vararg"-Ausdruck "..." auf diese Argumente zugreifen..
Im interaktiven Modus wartet der Interpreter bei unvollständigen Anweisungen auf deren Komplettierung durch die Anzeige einer anderen Aufforderung.
Wenn die globale Variable _PROMPT eine Zeichenkette enthält, dann wird deren Wert als Eingabeaufforderung verwendet. Wenn die globale Variable _PROMPT2 eine Zeichenkette enthält, wird deren Wert als alternative Eingabeaufforderung (bei nicht vollständigen Anweisungen) verwendet. Demnach können beide Eingabeaufforderungen direkt über die Kommandozeile oder in Lua-Programmen durch Zuweisung an _PROMPT verändert werden. Sehen Sie sich das nächste Beispiel an:
$ lua -e"_PROMPT='myprompt '" -i
(Die äußeren Anführungszeichen sind für die Konsole, die Inneren für Lua.) Beachten Sie die Benutzung von -i für den interaktiven Modus; andernfalls würde das Programm direkt nach der Zuweisung an _PROMPT enden.
Um Lua als Interpreter auf unixoiden Systemen nutzen zu können, überspringt der Interpreter die erste Zeile eines Datenblocks, wenn dieser mit # beginnt. Insofern können Lua-Skripte durch chmod +x und die #!-Form in ein ausführbares Programm überführt werden, so wie in
#!/usr/local/bin/lua
(Natürlich kann der Ort des Lua-Interpreters auf Ihrer Maschine abweichen. Wenn sich lua in Ihrem PATH befindet, ist
#!/usr/bin/env lua
eine komfortablere Lösung.)

7 Inkompatibilitäten zu vorherigen Versionen

Hier listen wir die Inkompatibilitäten auf, welchen Sie möglicherweise begegnen werden, wenn Sie ein Programm von Lua 5.0 auf 5.1 portieren. Die meisten der Inkompatibilitäten können Sie vermeiden, indem Sie Lua mit den entsprechenden Einstellungen kompilieren (s. Datei luaconf.h). Jedoch werden all diese Kompatibilitätsoptionen in der nächsten Version von Lua entfernt.

7.1 Änderungen der Sprache

7.2 Änderungen in den Bibliotheken

7.3 Änderungen der API

8 Die komplette Syntax von Lua

Hier ist die komplette Syntax von Lua in EBNF: (dies beschreibt nicht die Operatorpräzedenz)