Lua 5.0 Reference Manual

Last revised on November 25, 2003

Copyright © 2003 Tecgraf, PUC-Rio. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Copies of this manual can be obtained at Lua's official web site, `www.lua.org`.

The Lua logo was designed by A. Nakonechny. Copyright © 1998. All rights reserved.

Lua 5.0 Reference Manual

Roberto Ierusalimschy Luiz Henrique de Figueiredo Waldemar Celes

lua@tecgraf.puc-rio.br

Tecgraf — Computer Science Department — PUC-Rio

PUC-RioInf.MCC14/03 Abril 2003

Abstract

Lua is a powerful, light-weight programming language designed for extending applications. Lua is also frequently used as a general-purpose, stand-alone language. Lua combines simple procedural syntax (similar to Pascal) with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, interpreted from opcodes, and has automatic memory management with garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

This document describes version 5.0 of the Lua programming language and the Application Program Interface (API) that allows interaction between Lua programs and their host C programs.

Resumo

Lua é uma linguagem de programação poderosa e leve, projetada para estender aplicações. Lua também é frequentemente usada como uma linguagem de propósito geral. Lua combina programação procedural (com sintaxe semelhante à de Pascal) com poderosas construções para descrição de dados, baseadas em tabelas associativas e semântica extensível. Lua é tipada dinamicamente, interpretada a partir de *opcodes*, e tem gerenciamento automático de memória com coleta de lixo. Essas características fazem de Lua uma linguagem ideal para configuração, automação (*scripting*) e prototipagem rápida.

Este documento descreve a versão 5.0 da linguagem de programação Lua e a Interface de Programação (API) que permite a interação entre programas Lua e programas C hospedeiros.

Contents

1 Introduction **1**

2 The Language **1**
2.1 Lexical Conventions . . . 1
2.2 Values and Types . . . 3
2.2.1 Coercion . . . 4
2.3 Variables . . . 4
2.4 Statements . . . 4
2.4.1 Chunks . . . 5
2.4.2 Blocks . . . 5
2.4.3 Assignment . . . 5
2.4.4 Control Structures . . . 6
2.4.5 For Statement . . . 6
2.4.6 Function Calls as Statements . . . 8
2.4.7 Local Declarations . . . 8
2.5 Expressions . . . 8
2.5.1 Arithmetic Operators . . . 8
2.5.2 Relational Operators . . . 9
2.5.3 Logical Operators . . . 9
2.5.4 Concatenation . . . 10
2.5.5 Precedence . . . 10
2.5.6 Table Constructors . . . 10
2.5.7 Function Calls . . . 11
2.5.8 Function Definitions . . . 12
2.6 Visibility Rules . . . 14
2.7 Error Handling . . . 14
2.8 Metatables . . . 15
2.9 Garbage Collection . . . 20
2.9.1 Garbage-Collection Metamethods . . . 20
2.9.2 Weak Tables . . . 20
2.10 Coroutines . . . 21

3 The Application Program Interface **22**
3.1 States . . . 22
3.2 The Stack and Indices . . . 23
3.3 Stack Manipulation . . . 23
3.4 Querying the Stack . . . 24
3.5 Getting Values from the Stack . . . 25
3.6 Pushing Values onto the Stack . . . 26
3.7 Controlling Garbage Collection . . . 27
3.8 Userdata . . . 27
3.9 Metatables . . . 28
3.10 Loading Lua Chunks . . . 28
3.11 Manipulating Tables . . . 28
3.12 Manipulating Environments . . . 30
3.13 Using Tables as Arrays . . . 30

3.14 Calling Functions . . . 30
3.15 Protected Calls . . . 31
3.16 Defining C Functions . . . 32
3.17 Defining C Closures . . . 33
3.18 Registry . . . 33
3.19 Error Handling in C . . . 33
3.20 Threads . . . 34

4 The Debug Interface **35**
4.1 Stack and Function Information . . . 35
4.2 Manipulating Local Variables and Upvalues . . . 36
4.3 Hooks . . . 37

5 Standard Libraries **38**
5.1 Basic Functions . . . 39
5.2 Coroutine Manipulation . . . 43
5.3 String Manipulation . . . 44
5.4 Table Manipulation . . . 49
5.5 Mathematical Functions . . . 50
5.6 Input and Output Facilities . . . 51
5.7 Operating System Facilities . . . 53
5.8 The Reflexive Debug Interface . . . 55

6 Lua Stand-alone **57**

Incompatibilities with Previous Versions **58**

The Complete Syntax of Lua **60**

Index **61**

1 Introduction

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Lua is intended to be used as a powerful, light-weight configuration language for any program that needs one. Lua is implemented as a library, written in *clean* C (that is, in the common subset of ANSI C and C++).

Being an extension language, Lua has no notion of a "main" program: it only works *embedded* in a host client, called the *embedding program* or simply the *host*. This host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. Through the use of C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework.

The Lua distribution includes a stand-alone embedding program, `lua`, that uses the Lua library to offer a complete Lua interpreter.

Lua is free software, and is provided as usual with no guarantees, as stated in its copyright notice. The implementation described in this manual is available at Lua's official web site, `www.lua.org`.

Like any other reference manual, this document is dry in places. For a discussion of the decisions behind the design of Lua, see the papers below, which are available at Lua's web site.

- R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice & Experience* **26** #6 (1996) 635–652.
- L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. The design and implementation of a language for extending applications. *Proceedings of XXI Brazilian Seminar on Software and Hardware* (1994) 273–283.
- L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal* **21** #12 (Dec 1996) 26–33.
- R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of an extension language: a history of Lua, *Proceedings of V Brazilian Symposium on Programming Languages* (2001) B-14–B-28.

Lua means "moon" in Portuguese and is pronounced LOO-ah.

2 The Language

This section describes the lexis, the syntax, and the semantics of Lua. In other words, this section describes which tokens are valid, how they can be combined, and what their combinations mean.

The language constructs will be explained using the usual extended BNF, in which { *a* } means 0 or more *a*'s, and [*a*] means an optional *a*. Non-terminals are shown in *italics*, keywords are shown in **bold**, and other terminal symbols are shown in `typewriter` font, enclosed in single quotes.

2.1 Lexical Conventions

Identifiers in Lua can be any string of letters, digits, and underscores, not beginning with a digit. This coincides with the definition of identifiers in most languages. (The definition of letter depends on the current locale: any character considered alphabetic by the current locale can be used in an identifier.)

The following *keywords* are reserved and cannot be used as identifiers:

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

Lua is a case-sensitive language: `and` is a reserved word, but `And` and `AND` are two different, valid identifiers. As a convention, identifiers starting with an underscore followed by uppercase letters (such as `_VERSION`) are reserved for internal variables used by Lua.

The following strings denote other tokens:

```
+     -     *     /     ^     =
~=    <=    >=    <     >     ==
(     )     {     }     [     ]
;     :     ,     .     ..    ...
```

Literal strings can be delimited by matching single or double quotes, and can contain the following C-like escape sequences:

- `\a` — bell
- `\b` — backspace
- `\f` — form feed
- `\n` — newline
- `\r` — carriage return
- `\t` — horizontal tab
- `\v` — vertical tab
- `\\` — backslash
- `\"` — quotation mark
- `\'` — apostrophe
- `\[` — left square bracket
- `\]` — right square bracket

Moreover, a '`\`*newline*' (that is, a backslash followed by a real newline) results in a newline in the string. A character in a string may also be specified by its numerical value using the escape sequence '`\`*ddd*', where *ddd* is a sequence of up to three decimal digits. Strings in Lua may contain any 8-bit value, including embedded zeros, which can be specified as '`\0`'.

Literal strings can also be delimited by matching double square brackets `[[ ... ]]`. Literals in this bracketed form may run for several lines, may contain nested `[[ ... ]]` pairs, and do not interpret any escape sequences. For convenience, when the opening '`[[`' is immediately followed by a newline, the newline is not included in the string. As an example, in a system using ASCII (in which '`a`' is coded as 97, newline is coded as 10, and '`1`' is coded as 49), the four literals below denote the same string:

```
(1)   "alo\n123\""
(2)   '\97lo\10\04923"'
(3)   [[alo
      123"]]
(4)   [[
      alo
      123"]]
```

Numerical constants may be written with an optional decimal part and an optional decimal exponent. Examples of valid numerical constants are

```
3     3.0     3.1416  314.16e-2   0.31416E1
```

Comments start anywhere outside a string with a double hyphen (`--`). If the text immediately after `--` is different from `[[`, the comment is a *short comment*, which runs until the end of the line. Otherwise, it is a *long comment*, which runs until the corresponding `]]`. Long comments may run for several lines and may contain nested `[[ ... ]]` pairs.

For convenience, the first line of a chunk is skipped if it starts with `#`. This facility allows the use of Lua as a script interpreter in Unix systems (see §6).

2.2 Values and Types

Lua is a *dynamically typed language*. That means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

There are eight basic types in Lua: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread*, and *table*. *Nil* is the type of the value **nil**, whose main property is to be different from any other value; usually it represents the absence of a useful value. *Boolean* is the type of the values **false** and **true**. In Lua, both **nil** and **false** make a condition false; any other value makes it true. *Number* represents real (double-precision floating-point) numbers. (It is easy to build Lua interpreters that use other internal representations for numbers, such as single-precision float or long integers.) *String* represents arrays of characters. Lua is 8-bit clean: Strings may contain any 8-bit character, including embedded zeros (`'\0'`) (see §2.1).

Functions are *first-class values* in Lua. That means that functions can be stored in variables, passed as arguments to other functions, and returned as results. Lua can call (and manipulate) functions written in Lua and functions written in C (see §2.5.7).

The type *userdata* is provided to allow arbitrary C data to be stored in Lua variables. This type corresponds to a block of raw memory and has no pre-defined operations in Lua, except assignment and identity test. However, by using *metatables*, the programmer can define operations for userdata values (see §2.8). Userdata values cannot be created or modified in Lua, only through the C API. This guarantees the integrity of data owned by the host program.

The type *thread* represents independent threads of execution and it is used to implement coroutines.

The type *table* implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any value (except **nil**). Moreover, tables can be *heterogeneous*, that is, they can contain values of all types (except **nil**). Tables are the sole data structuring mechanism in Lua; they may be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc. To represent records, Lua uses the field name as an index. The language supports this representation by providing `a.name` as syntactic sugar for `a["name"]`. There are several convenient ways to create tables in Lua (see §2.5.6).

Like indices, the value of a table field can be of any type (except **nil**). In particular, because functions are first class values, table fields may contain functions. Thus tables may also carry *methods* (see §2.5.8).

Tables, functions, and userdata values are *objects*: variables do not actually *contain* these values, only *references* to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy.

The library function `type` returns a string describing the type of a given value (see §5.1).

2.2.1 Coercion

Lua provides automatic conversion between string and number values at run time. Any arithmetic operation applied to a string tries to convert that string to a number, following the usual rules. Conversely, whenever a number is used where a string is expected, the number is converted to a string, in a reasonable format. For complete control of how numbers are converted to strings, use the `format` function from the string library (see §5.3).

2.3 Variables

Variables are places that store values. There are three kinds of variables in Lua: global variables, local variables, and table fields.

A single name can denote a global variable or a local variable (or a formal parameter of a function, which is a particular form of local variable):

var → `Name`

Variables are assumed to be global unless explicitly declared local (see §2.4.7). Local variables are *lexically scoped*: Local variables can be freely accessed by functions defined inside their scope (see §2.6).

Before the first assignment to a variable, its value is **nil**.

Square brackets are used to index a table:

var → *prefixexp* '`[`' *exp* '`]`'

The first expression (*prefixexp*)should result in a table value; the second expression (*exp*) identifies a specific entry inside that table. The expression denoting the table to be indexed has a restricted syntax; see §2.5 for details.

The syntax `var.NAME` is just syntactic sugar for `var["NAME"]`:

var → *prefixexp* '`.`' `Name`

The meaning of accesses to global variables and table fields can be changed via metatables. An access to an indexed variable `t[i]` is equivalent to a call `gettable_event(t,i)`. (See §2.8 for a complete description of the `gettable_event` function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)

All global variables live as fields in ordinary Lua tables, called *environment tables* or simply *environments*. Functions written in C and exported to Lua (*C functions*) all share a common *global environment*. Each function written in Lua (a *Lua function*) has its own reference to an environment, so that all global variables in that function will refer to that environment table. When a function is created, it inherits the environment from the function that created it. To change or get the environment table of a Lua function, you call `setfenv` or `getfenv` (see §5.1).

An access to a global variable `x` is equivalent to `_env.x`, which in turn is equivalent to

```
gettable_event(_env, "x")
```

where `_env` is the environment of the running function. (The `_env` variable is not defined in Lua. We use it here only for explanatory purposes.)

2.4 Statements

Lua supports an almost conventional set of statements, similar to those in Pascal or C. This set includes assignment, control structures, procedure calls, table constructors, and variable declarations.

2.4.1 Chunks

The unit of execution of Lua is called a *chunk*. A chunk is simply a sequence of statements, which are executed sequentially. Each statement can be optionally followed by a semicolon:

chunk → { *stat* [';'] }

Lua handles a chunk as the body of an anonymous function (see §2.5.8). As such, chunks can define local variables and return values.

A chunk may be stored in a file or in a string inside the host program. When a chunk is executed, first it is pre-compiled into opcodes for a virtual machine, and then the compiled code is executed by an interpreter for the virtual machine.

Chunks may also be pre-compiled into binary form; see program `luac` for details. Programs in source and compiled forms are interchangeable; Lua automatically detects the file type and acts accordingly.

2.4.2 Blocks

A block is a list of statements; syntactically, a block is equal to a chunk:

block → *chunk*

A block may be explicitly delimited to produce a single statement:

stat → **do** *block* **end**

Explicit blocks are useful to control the scope of variable declarations. Explicit blocks are also sometimes used to add a **return** or **break** statement in the middle of another block (see §2.4.4).

2.4.3 Assignment

Lua allows multiple assignment. Therefore, the syntax for assignment defines a list of variables on the left side and a list of expressions on the right side. The elements in both lists are separated by commas:

stat → *varlist1* '=' *explist1*
varlist1 → *var* { ',' *var* }
explist1 → *exp* { ',' *exp* }

Expressions are discussed in §2.5.

Before the assignment, the list of values is *adjusted* to the length of the list of variables. If there are more values than needed, the excess values are thrown away. If there are fewer values than needed, the list is extended with as many **nil**'s as needed. If the list of expressions ends with a function call, then all values returned by that function call enter in the list of values, before the adjustment (except when the call is enclosed in parentheses; see §2.5).

The assignment statement first evaluates all its expressions and only then are the assignments performed. Thus the code

```
i = 3
i, a[i] = i+1, 20
```

sets `a[3]` to 20, without affecting `a[4]` because the `i` in `a[i]` is evaluated (to 3) before it is assigned 4. Similarly, the line

```
x, y = y, x
```

exchanges the values of `x` and `y`.

The meaning of assignments to global variables and table fields can be changed via metatables. An assignment to an indexed variable `t[i] = val` is equivalent to `settable_event(t,i,val)`. (See §2.8 for a complete description of the `settable_event` function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)

An assignment to a global variable `x = val` is equivalent to the assignment `_env.x = val`, which in turn is equivalent to

```
settable_event(_env, "x", val)
```

where `_env` is the environment of the running function. (The `_env` variable is not defined in Lua. We use it here only for explanatory purposes.)

2.4.4 Control Structures

The control structures **if**, **while**, and **repeat** have the usual meaning and familiar 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 also has a **for** statement, in two flavors (see §2.4.5).

The condition expression *exp* of a control structure may return any value. Both **false** and **nil** are considered false. All values different from **nil** and **false** are considered true (in particular, the number 0 and the empty string are also true).

The **return** statement is used to return values from a function or from a chunk. Functions and chunks may return more than one value, so the syntax for the **return** statement is

stat → **return** [*explist1*]

The **break** statement can be used to terminate the execution of a **while**, **repeat**, or **for** loop, skipping to the next statement after the loop:

stat → **break**

A **break** ends the innermost enclosing loop.

For syntactic reasons, **return** and **break** statements can only be written as the *last* statement of a block. If it is really necessary to **return** or **break** in the middle of a block, then an explicit inner block can be used, as in the idioms '`do return end`' and '`do break end`', because now **return** and **break** are the last statements in their (inner) blocks. In practice, those idioms are only used during debugging.

2.4.5 For Statement

The **for** statement has two forms: one numeric and one generic.

The numeric **for** loop repeats a block of code while a control variable runs through an arithmetic progression. It has the following syntax:

stat → **for** `Name` '`=`' *exp* '`,`' *exp* ['`,`' *exp*] **do** *block* **end**

The *block* is repeated for *name* starting at the value of the first *exp*, until it passes the second *exp* by steps of the third *exp*. More precisely, a **for** statement like

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

is equivalent to the code:

```
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) or (_step<=0 and var>=_limit) do
    block
    var = var + _step
  end
end
```

Note the following:

- All three control expressions are evaluated only once, before the loop starts. They must all result in numbers.
- `_limit` and `_step` are invisible variables. The names are here for explanatory purposes only.
- The behavior is *undefined* if you assign to `var` inside the block.
- If the third expression (the step) is absent, then a step of 1 is used.
- You can use **break** to exit a **for** loop.
- The loop variable `var` is local to the statement; you cannot use its value after the **for** ends or is broken. If you need the value of the loop variable `var`, then assign it to another variable before breaking or exiting the loop.

The generic **for** statement works over functions, called *iterators*. For each iteration, it calls its iterator function to produce a new value, stopping when the new value is **nil**. The generic **for** loop has the following syntax:

stat → **for** `Name` { ‘`,`’ `Name` } **in** *explist1* **do** *block* **end**

A **for** statement like

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

is equivalent to the code:

```
do
  local _f, _s, var_1 = explist
  local var_2, ... , var_n
  while true do
    var_1, ..., var_n = _f(_s, var_1)
    if var_1 == nil then break end
    block
  end
end
```

Note the following:

- `explist` is evaluated only once. Its results are an *iterator* function, a *state*, and an initial value for the first *iterator variable*.
- `_f` and `_s` are invisible variables. The names are here for explanatory purposes only.

- The behavior is *undefined* if you assign to `var_1` inside the block.
- You can use **break** to exit a **for** loop.
- The loop variables `var_i` are local to the statement; you cannot use their values after the **for** ends. If you need these values, then assign them to other variables before breaking or exiting the loop.

2.4.6 Function Calls as Statements

To allow possible side-effects, function calls can be executed as statements:

stat → *functioncall*

In this case, all returned values are thrown away. Function calls are explained in §2.5.7.

2.4.7 Local Declarations

Local variables may be declared anywhere inside a block. The declaration may include an initial assignment:

stat → **local** *namelist* ['`=`' *explist1*]
namelist → `Name` { '`,`' `Name` }

If present, an initial assignment has the same semantics of a multiple assignment (see §2.4.3). Otherwise, all variables are initialized with **nil**.

A chunk is also a block (see §2.4.1), so local variables can be declared in a chunk outside any explicit block. Such local variables die when the chunk ends.

The visibility rules for local variables are explained in §2.6.

2.5 Expressions

The basic expressions in Lua are the following:

exp → *prefixexp*
exp → **nil** | **false** | **true**
exp → `Number`
exp → `Literal`
exp → *function*
exp → *tableconstructor*
prefixexp → *var* | *functioncall* | '`(`' *exp* '`)`'

Numbers and literal strings are explained in §2.1; variables are explained in §2.3; function definitions are explained in §2.5.8; function calls are explained in §2.5.7; table constructors are explained in §2.5.6.

An expression enclosed in parentheses always results in only one value. Thus, `(f(x,y,z))` is always a single value, even if `f` returns several values. (The value of `(f(x,y,z))` is the first value returned by `f` or **nil** if `f` does not return any values.)

Expressions can also be built with arithmetic operators, relational operators, and logical operators, all of which are explained below.

2.5.1 Arithmetic Operators

Lua supports the usual arithmetic operators: the binary `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `^` (exponentiation); and unary `-` (negation). If the operands are numbers,

or strings that can be converted to numbers (see §2.2.1), then all operations except exponentiation have the usual meaning. Exponentiation calls a global function `__pow`; otherwise, an appropriate metamethod is called (see §2.8). The standard mathematical library defines function `__pow`, giving the expected meaning to exponentiation (see §5.5).

2.5.2 Relational Operators

The relational operators in Lua are

```
==    ~=    <     >     <=    >=
```

These operators always result in **false** or **true**.

Equality (`==`) first compares the type of its operands. If the types are different, then the result is **false**. Otherwise, the values of the operands are compared. Numbers and strings are compared in the usual way. Objects (tables, userdata, threads, and functions) are compared by *reference*: Two objects are considered equal only if they are the *same* object. Every time you create a new object (a table, userdata, or function), this new object is different from any previously existing object.

You can change the way that Lua compares tables and userdata using the "eq" metamethod (see §2.8).

The conversion rules of §2.2.1 *do not* apply to equality comparisons. Thus, `"0"==0` evaluates to **false**, and `t[0]` and `t["0"]` denote different entries in a table.

The operator `~=` is exactly the negation of equality (`==`).

The order operators work as follows. If both arguments are numbers, then they are compared as such. Otherwise, if both arguments are strings, then their values are compared according to the current locale. Otherwise, Lua tries to call the "lt" or the "le" metamethod (see §2.8).

2.5.3 Logical Operators

The logical operators in Lua are

```
and   or    not
```

Like the control structures (see §2.4.4), all logical operators consider both **false** and **nil** as false and anything else as true.

The operator **not** always return **false** or **true**.

The conjunction operator **and** returns its first argument if this value is **false** or **nil**; otherwise, **and** returns its second argument. The disjunction operator **or** returns its first argument if this value is different from **nil** and **false**; otherwise, **or** returns its second argument. Both **and** and **or** use short-cut evaluation, that is, the second operand is evaluated only if necessary. For example,

```
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
```

2.5.4 Concatenation

The string concatenation operator in Lua is denoted by two dots ('`..`'). If both operands are strings or numbers, then they are converted to strings according to the rules mentioned in §2.2.1. Otherwise, the "concat" metamethod is called (see §2.8).

2.5.5 Precedence

Operator precedence in Lua follows the table below, from lower to higher priority:

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /
not   - (unary)
^
```

You can use parentheses to change the precedences in an expression. The concatenation ('`..`') and exponentiation ('`^`') operators are right associative. All other binary operators are left associative.

2.5.6 Table Constructors

Table constructors are expressions that create tables. Every time a constructor is evaluated, a new table is created. Constructors can be used to create empty tables, or to create a table and initialize some of its fields. The general syntax for constructors is

tableconstructor → '{' [*fieldlist*] '}'
fieldlist → *field* { *fieldsep* *field* } [*fieldsep*]
field → '`[`' *exp* '`]`' '`=`' *exp* | `Name` '`=`' *exp* | *exp*
fieldsep → '`,`' | '`;`'

Each field of the form `[exp1] = exp2` adds to the new table an entry with key `exp1` and value `exp2`. A field of the form `name = exp` is equivalent to `["name"] = exp`. Finally, fields of the form `exp` are equivalent to `[i] = exp`, where `i` are consecutive numerical integers, starting with 1. Fields in the other formats do not affect this counting. For example,

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

is equivalent to

```
do
  local temp = {}
  temp[f(1)] = g
  temp[1] = "x"         -- 1st exp
  temp[2] = "y"         -- 2nd exp
  temp.x = 1            -- temp["x"] = 1
  temp[3] = f(x)        -- 3rd exp
  temp[30] = 23
  temp[4] = 45          -- 4th exp
  a = temp
end
```

If the last field in the list has the form `exp` and the expression is a function call, then all values returned by the call enter the list consecutively (see §2.5.7). To avoid this, enclose the function call in parentheses (see §2.5).

The field list may have an optional trailing separator, as a convenience for machine-generated code.

2.5.7 Function Calls

A function call in Lua has the following syntax:

functioncall → *prefixexp args*

In a function call, first *prefixexp* and *args* are evaluated. If the value of *prefixexp* has type *function*, then that function is called with the given arguments. Otherwise, its "call" metamethod is called, having as first parameter the value of *prefixexp*, followed by the original call arguments (see §2.8).

The form

functioncall → *prefixexp* ':' `Name` *args*

can be used to call "methods". A call `v:name(...)` is syntactic sugar for `v.name(v,...)`, except that `v` is evaluated only once.

Arguments have the following syntax:

args → '(' [*explist1*] ')'
args → *tableconstructor*
args → `Literal`

All argument expressions are evaluated before the call. A call of the form `f{...}` is syntactic sugar for `f({...})`, that is, the argument list is a single new table. A call of the form `f'...'` (or `f"..."` or `f[[...]]`) is syntactic sugar for `f('...')`, that is, the argument list is a single literal string.

Because a function can return any number of results (see §2.4.4), the number of results must be adjusted before they are used. If the function is called as a statement (see §2.4.6), then its return list is adjusted to zero elements, thus discarding all returned values. If the function is called inside another expression or in the middle of a list of expressions, then its return list is adjusted to one element, thus discarding all returned values except the first one. If the function is called as the last element of a list of expressions, then no adjustment is made (unless the call is enclosed in parentheses).

Here are some examples:

```
f()                -- adjusted to 0 results
g(f(), x)          -- f() is adjusted to 1 result
g(x, f())          -- g gets x plus all values returned by f()
a,b,c = f(), x     -- f() is adjusted to 1 result (and c gets nil)
a,b,c = x, f()     -- f() is adjusted to 2 results
a,b,c = f()        -- f() is adjusted to 3 results
return f()         -- returns all values returned by f()
return x,y,f()     -- returns x, y, and all values returned by f()
{f()}              -- creates a list with all values returned by f()
{f(), nil}         -- f() is adjusted to 1 result
```

If you enclose a function call in parentheses, then it is adjusted to return exactly one value:

```
return x,y,(f())   -- returns x, y, and the first value from f()
{(f())}            -- creates a table with exactly one element
```

As an exception to the free-format syntax of Lua, you cannot put a line break before the '`(`' in a function call. That restriction avoids some ambiguities in the language. If you write

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

Lua would read that as `a = f(g).x(a)`. So, if you want two statements, you must add a semi-colon between them. If you actually want to call `f`, you must remove the line break before `(g)`.

A call of the form `return` *functioncall* is called a *tail call*. Lua implements *proper tail calls* (or *proper tail recursion*): In a tail call, the called function reuses the stack entry of the calling function. Therefore, there is no limit on the number of nested tail calls that a program can execute. However, a tail call erases any debug information about the calling function. Note that a tail call only happens with a particular syntax, where the **return** has one single function call as argument; this syntax makes the calling function returns exactly the returns of the called function. So, all the following examples are not tails calls:

```
return (f(x))        -- results adjusted to 1
return 2 * f(x)
return x, f(x)       -- additional results
f(x); return         -- results discarded
return x or f(x)     -- results adjusted to 1
```

2.5.8 Function Definitions

The syntax for function definition is

function → **function** *funcbody*
funcbody → '`(`' [*parlist1*] '`)`' *block* **end**

The following syntactic sugar simplifies function definitions:

stat → **function** *funcname funcbody*
stat → **local function** `Name` *funcbody*
funcname → `Name` { '`.`' `Name` } ['`:`' `Name`]

The statement

```
function f () ... end
```

translates to

```
f = function () ... end
```

The statement

```
function t.a.b.c.f () ... end
```

translates to

```
t.a.b.c.f = function () ... end
```

The statement

```
local function f () ... end
```

translates to

```
local f; f = function () ... end
```

A function definition is an executable expression, whose value has type *function*. When Lua pre-compiles a chunk, all its function bodies are pre-compiled too. Then, whenever Lua executes the function definition, the function is *instantiated* (or *closed*). This function instance (or *closure*) is the final value of the expression. Different instances of the same function may refer to different external local variables and may have different environment tables.

Parameters act as local variables that are initialized with the argument values:

parlist1 → *namelist* [',' '...']
parlist1 → '...'

When a function is called, the list of arguments is adjusted to the length of the list of parameters, unless the function is a variadic or *vararg function*, which is indicated by three dots ('...') at the end of its parameter list. A vararg function does not adjust its argument list; instead, it collects all extra arguments into an implicit parameter, called `arg`. The value of `arg` is a table, with a field `n` that holds the number of extra arguments and with the extra arguments at positions 1, 2, ..., `n`.

As an example, consider the following definitions:

```
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
```

Then, we have the following mapping from arguments to parameters:

```
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, arg={n=0}
g(3, 4)          a=3, b=4,   arg={n=0}
g(3, 4, 5, 8)    a=3, b=4,   arg={5, 8; n=2}
g(5, r())        a=5, b=1,   arg={2, 3; n=2}
```

Results are returned using the **return** statement (see §2.4.4). If control reaches the end of a function without encountering a **return** statement, then the function returns with no results.

The *colon* syntax is used for defining *methods*, that is, functions that have an implicit extra parameter `self`. Thus, the statement

```
function t.a.b.c:f (...) ... end
```

is syntactic sugar for

```
t.a.b.c.f = function (self, ...) ... end
```

2.6 Visibility Rules

Lua is a lexically scoped language. The scope of variables begins at the first statement *after* their declaration and lasts until the end of the innermost block that includes the declaration. For instance:

```
x = 10                -- global variable
do                    -- new block
  local x = x         -- new 'x', with value 10
  print(x)            --> 10
  x = x+1
  do                  -- another block
    local x = x+1     -- another 'x'
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10  (the global one)
```

Notice that, in a declaration like `local x = x`, the new `x` being declared is not in scope yet, and so the second `x` refers to the outside variable.

Because of the lexical scoping rules, local variables can be freely accessed by functions defined inside their scope. For instance:

```
local counter = 0
function inc (x)
  counter = counter + x
  return counter
end
```

A local variable used by an inner function is called an *upvalue*, or *external local variable*, inside the inner function.

Notice that each execution of a **local** statement defines new local variables. Consider the following example:

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

The loop creates ten closures (that is, ten instances of the anonymous function). Each of these closures uses a different `y` variable, while all of them share the same `x`.

2.7 Error Handling

Because Lua is an extension language, all Lua actions start from C code in the host program calling a function from the Lua library (see §3.15). Whenever an error occurs during Lua compilation or execution, control returns to C, which can take appropriate measures (such as print an error message).

Lua code can explicitly generate an error by calling the `error` function (see §5.1). If you need to catch errors in Lua, you can use the `pcall` function (see §5.1).

2.8 Metatables

Every table and userdata object in Lua may have a *metatable*. This *metatable* is an ordinary Lua table that defines the behavior of the original table and userdata under certain special operations. You can change several aspects of the behavior of an object by setting specific fields in its metatable. For instance, when an object is the operand of an addition, Lua checks for a function in the field `"__add"` in its metatable. If it finds one, Lua calls that function to perform the addition.

We call the keys in a metatable *events* and the values *metamethods*. In the previous example, the event is `"add"` and the metamethod is the function that performs the addition.

You can query and change the metatable of an object through the `set/getmetatable` functions (see §5.1).

A metatable may control how an object behaves in arithmetic operations, order comparisons, concatenation, and indexing. A metatable can also define a function to be called when a userdata is garbage collected. For each of those operations Lua associates a specific key called an *event*. When Lua performs one of those operations over a table or a userdata, it checks whether that object has a metatable with the corresponding event. If so, the value associated with that key (the *metamethod*) controls how Lua will perform the operation.

Metatables control the operations listed next. Each operation is identified by its corresponding name. The key for each operation is a string with its name prefixed by two underscores; for instance, the key for operation "add" is the string `"__add"`. The semantics of these operations is better explained by a Lua function describing how the interpreter executes that operation.

The code shown here in Lua is only illustrative; the real behavior is hard coded in the interpreter and it is much more efficient than this simulation. All functions used in these descriptions (`rawget`, `tonumber`, etc.) are described in §5.1. In particular, to retrieve the metamethod of a given object, we use the expression

```
metatable(obj)[event]
```

This should be read as

```
rawget(metatable(obj) or {}, event)
```

That is, the access to a metamethod does not invoke other metamethods, and the access to objects with no metatables does not fail (it simply results in **nil**).

"add": the `+` operation.

The function `getbinhandler` below defines how Lua chooses a handler for a binary operation. First, Lua tries the first operand. If its type does not define a handler for the operation, then Lua tries the second operand.

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

Using that function, the behavior of the `op1 + op2` is

```
function add_event (op1, op2)
  local o1, o2 = tonumber(op1), tonumber(op2)
  if o1 and o2 then  -- both operands are numeric?
    return o1 + o2   -- '+' here is the primitive 'add'
  else  -- at least one of the operands is not numeric
    local h = getbinhandler(op1, op2, "__add")
    if h then
      -- call the handler with both operands
      return h(op1, op2)
    else  -- no handler available: default behavior
      error("...")
    end
  end
end
```

"sub": the - operation. Behavior similar to the "add" operation.
"mul": the * operation. Behavior similar to the "add" operation.
"div": the / operation. Behavior similar to the "add" operation.
"pow": the ^ (exponentiation) operation.

```
function pow_event (op1, op2)
  local o1, o2 = tonumber(op1), tonumber(op2)
  if o1 and o2 then  -- both operands are numeric?
    return __pow(o1, o2)   -- call global '__pow'
  else  -- at least one of the operands is not numeric
    local h = getbinhandler(op1, op2, "__pow")
    if h then
      -- call the handler with both operands
      return h(op1, op2)
    else  -- no handler available: default behavior
      error("...")
    end
  end
 end
```

"unm": the unary - operation.

```
function unm_event (op)
  local o = tonumber(op)
  if o then  -- operand is numeric?
    return -o  -- '-' here is the primitive 'unm'
  else  -- the operand is not numeric.
    -- Try to get a handler from the operand
    local h = metatable(op).__unm
    if h then
      -- call the handler with the operand and nil
      return h(op, nil)
```

```
    else  -- no handler available: default behavior
      error("...")
    end
  end
end
```

"concat": the `..` (concatenation) operation.

```
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 string concatenation
  else
    local h = getbinhandler(op1, op2, "__concat")
    if h then
      return h(op1, op2)
    else
      error("...")
    end
  end
end
```

"eq": the `==` operation. The function `getcomphandler` defines how Lua chooses a metamethod for comparison operators. A metamethod only is selected when both objects being compared have the same type and the same metamethod for the selected operation.

```
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
```

The "eq" event is defined as follows:

```
function eq_event (op1, op2)
  if type(op1) ~= type(op2) then  -- different types?
    return false   -- different objects
  end
  if op1 == op2 then   -- primitive equal?
    return true   -- objects are equal
  end
  -- try metamethod
  local h = getcomphandler(op1, op2, "__eq")
  if h then
    return h(op1, op2)
  else
    return false
```

```
    end
  end
```

`a ~= b` is equivalent to `not (a == b)`.

"lt": the `<` operation.

```
  function lt_event (op1, op2)
    if type(op1) == "number" and type(op2) == "number" then
      return op1 < op2   -- numeric comparison
    elseif type(op1) == "string" and type(op2) == "string" then
      return op1 < op2   -- lexicographic comparison
    else
      local h = getcomphandler(op1, op2, "__lt")
      if h then
        return h(op1, op2)
      else
        error("...");
      end
    end
  end
```

`a > b` is equivalent to `b < a`.

"le": the `<=` operation.

```
  function le_event (op1, op2)
    if type(op1) == "number" and type(op2) == "number" then
      return op1 <= op2   -- numeric comparison
    elseif type(op1) == "string" and type(op2) == "string" then
      return op1 <= op2   -- lexicographic comparison
    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` is equivalent to `b <= a`. Note that, in the absence of a "le" metamethod, Lua tries the "lt", assuming that `a <= b` is equivalent to `not (b < a)`.

"index": The indexing access `table[key]`.

```
function gettable_event (table, key)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    if v ~= nil then return v end
    h = metatable(table).__index
    if h == nil then return nil end
  else
    h = metatable(table).__index
    if h == nil then
      error("...");
    end
  end
  if type(h) == "function" then
    return h(table, key)      -- call the handler
  else return h[key]          -- or repeat operation on it
end
```

"newindex": The indexing assignment `table[key] = value`.

```
function settable_event (table, key, value)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    if v ~= nil then rawset(table, key, value); return end
    h = metatable(table).__newindex
    if h == nil then rawset(table, key, value); return end
  else
    h = metatable(table).__newindex
    if h == nil then
      error("...");
    end
  end
  if type(h) == "function" then
    return h(table, key,value)    -- call the handler
  else h[key] = value             -- or repeat operation on it
end
```

"call": called when Lua calls a value.

```
function function_event (func, ...)
  if type(func) == "function" then
    return func(unpack(arg))   -- primitive call
  else
    local h = metatable(func).__call
    if h then
      return h(func, unpack(arg))
    else
```

```
        error("...")
      end
    end
  end
```

2.9 Garbage Collection

Lua does automatic memory management. That means that you do not have to worry about allocating memory for new objects and freeing it when the objects are no longer needed. Lua manages memory automatically by running a *garbage collector* from time to time to collect all *dead objects* (that is, those objects that are no longer accessible from Lua). All objects in Lua are subject to automatic management: tables, userdata, functions, threads, and strings.

Lua uses two numbers to control its garbage-collection cycles. One number counts how many bytes of dynamic memory Lua is using; the other is a threshold. When the number of bytes crosses the threshold, Lua runs the garbage collector, which reclaims the memory of all dead objects. The byte counter is adjusted, and then the threshold is reset to twice the new value of the byte counter.

Through the C API, you can query those numbers and change the threshold (see §3.7). Setting the threshold to zero actually forces an immediate garbage-collection cycle, while setting it to a huge number effectively stops the garbage collector. Using Lua code you have a more limited control over garbage-collection cycles, through the `gcinfo` and `collectgarbage` functions (see §5.1).

2.9.1 Garbage-Collection Metamethods

Using the C API, you can set garbage-collector metamethods for userdata (see §2.8). These metamethods are also called *finalizers*. Finalizers allow you to coordinate Lua's garbage collection with external resource management (such as closing files, network or database connections, or freeing your own memory).

Free userdata with a field `__gc` in their metatables are not collected immediately by the garbage collector. Instead, Lua puts them in a list. After the collection, Lua does the equivalent of the following function for each userdata in that list:

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

At the end of each garbage-collection cycle, the finalizers for userdata are called in *reverse* order of their creation, among those collected in that cycle. That is, the first finalizer to be called is the one associated with the userdata created last in the program.

2.9.2 Weak Tables

A *weak table* is a table whose elements are *weak references*. A weak reference is ignored by the garbage collector. In other words, if the only references to an object are weak references, then the garbage collector will collect that object.

A weak table can have weak keys, weak values, or both. A table with weak keys allows the collection of its keys, but prevents the collection of its values. A table with both weak keys and

weak values allows the collection of both keys and values. In any case, if either the key or the value is collected, the whole pair is removed from the table. The weakness of a table is controlled by the value of the `__mode` field of its metatable. If the `__mode` field is a string containing the character '`k`', the keys in the table are weak. If `__mode` contains '`v`', the values in the table are weak.

After you use a table as a metatable, you should not change the value of its field `__mode`. Otherwise, the weak behavior of the tables controlled by this metatable is undefined.

2.10 Coroutines

Lua supports coroutines, also called *semi-coroutines* or *collaborative multithreading*. A coroutine in Lua represents an independent thread of execution. Unlike threads in multithread systems, however, a coroutine only suspends its execution by explicitly calling a yield function.

You create a coroutine with a call to `coroutine.create`. Its sole argument is a function that is the main function of the coroutine. The `create` function only creates a new coroutine and returns a handle to it (an object of type *thread*); it does not start the coroutine execution.

When you first call `coroutine.resume`, passing as its first argument the thread returned by `coroutine.create`, the coroutine starts its execution, at the first line of its main function. Extra arguments passed to `coroutine.resume` are given as parameters for the coroutine main function. After the coroutine starts running, it runs until it terminates or *yields*.

A coroutine can terminate its execution in two ways: Normally, when its main function returns (explicitly or implicitly, after the last instruction); and abnormally, if there is an unprotected error. In the first case, `coroutine.resume` returns **true**, plus any values returned by the coroutine main function. In case of errors, `coroutine.resume` returns **false** plus an error message.

A coroutine yields by calling `coroutine.yield`. When a coroutine yields, the corresponding `coroutine.resume` returns immediately, even if the yield happens inside nested function calls (that is, not in the main function, but in a function directly or indirectly called by the main function). In the case of a yield, `coroutine.resume` also returns **true**, plus any values passed to `coroutine.yield`. The next time you resume the same coroutine, it continues its execution from the point where it yielded, with the call to `coroutine.yield` returning any extra arguments passed to `coroutine.resume`.

The `coroutine.wrap` function creates a coroutine like `coroutine.create`, but instead of returning the coroutine itself, it returns a function that, when called, resumes the coroutine. Any arguments passed to that function go as extra arguments to resume. The function returns all the values returned by resume, except the first one (the boolean error code). Unlike `coroutine.resume`, this function does not catch errors; any error is propagated to the caller.

As an example, consider the next code:

```
function foo1 (a)
  print("foo", a)
  return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
      print("co-body", a, b)
      local r = foo1(a+1)
      print("co-body", r)
      local r, s = coroutine.yield(a+b, a-b)
      print("co-body", r, s)
```

```
      return b, "end"
end)

a, b = coroutine.resume(co, 1, 10)
print("main", a, b)
a, b, c = coroutine.resume(co, "r")
print("main", a, b, c)
a, b, c = coroutine.resume(co, "x", "y")
print("main", a, b, c)
a, b = coroutine.resume(co, "x", "y")
print("main", a, b)
```

When you run it, it produces the following output:

```
co-body 1       10
foo     2
main    true    4
co-body r
main    true    11      -9
co-body x       y
main    true    10      end
main    false   cannot resume dead coroutine
```

3 The Application Program Interface

This section describes the C API for Lua, that is, the set of C functions available to the host program to communicate with Lua. All API functions and related types and constants are declared in the header file `lua.h`.

Even when we use the term "function", any facility in the API may be provided as a *macro* instead. All such macros use each of its arguments exactly once (except for the first argument, which is always a Lua state), and so do not generate hidden side-effects.

3.1 States

The Lua library is fully reentrant: it has no global variables. The whole state of the Lua interpreter (global variables, stack, etc.) is stored in a dynamically allocated structure of type `lua_State`. A pointer to this state must be passed as the first argument to every function in the library, except to `lua_open`, which creates a Lua state from scratch.

Before calling any API function, you must create a state by calling `lua_open`:

```
lua_State *lua_open (void);
```

To release a state created with `lua_open`, call `lua_close`:

```
void lua_close (lua_State *L);
```

This function destroys all objects in the given Lua state (calling the corresponding garbage-collection metamethods, if any) and frees all dynamic memory used by that state. On several platforms, you may not need to call this function, because all resources are naturally released when the host program ends. On the other hand, long-running programs, such as a daemon or a web server, might need to release states as soon as they are not needed, to avoid growing too large.

3.2 The Stack and Indices

Lua uses a *virtual stack* to pass values to and from C. Each element in this stack represents a Lua value (**nil**, number, string, etc.).

Whenever Lua calls C, the called function gets a new stack, which is independent of previous stacks and of stacks of C functions that are still active. That stack initially contains any arguments to the C function, and it is where the C function pushes its results to be returned to the caller (see §3.16).

For convenience, most query operations in the API do not follow a strict stack discipline. Instead, they can refer to any element in the stack by using an *index*: A positive index represents an *absolute* stack position (starting at 1); a negative index represents an *offset* from the top of the stack. More specifically, if the stack has n elements, then index 1 represents the first element (that is, the element that was pushed onto the stack first) and index n represents the last element; index -1 also represents the last element (that is, the element at the top) and index $-n$ represents the first element. We say that an index is *valid* if it lies between 1 and the stack top (that is, if `1 <= abs(index) <= top`).

At any time, you can get the index of the top element by calling `lua_gettop`:

```
int lua_gettop (lua_State *L);
```

Because indices start at 1, the result of `lua_gettop` is equal to the number of elements in the stack (and so 0 means an empty stack).

When you interact with Lua API, *you are responsible for controlling stack overflow.* The function

```
int lua_checkstack (lua_State *L, int extra);
```

grows the stack size to `top + extra` elements; it returns false if it cannot grow the stack to that size. This function never shrinks the stack; if the stack is already larger than the new size, it is left unchanged.

Whenever Lua calls C, it ensures that at least `LUA_MINSTACK` stack positions are available. `LUA_MINSTACK` is defined in `lua.h` as 20, so that usually you do not have to worry about stack space unless your code has loops pushing elements onto the stack.

Most query functions accept as indices any value inside the available stack space, that is, indices up to the maximum stack size you have set through `lua_checkstack`. Such indices are called *acceptable indices*. More formally, we define an *acceptable index* as follows:

```
(index < 0 && abs(index) <= top) || (index > 0 && index <= stackspace)
```

Note that 0 is never an acceptable index.

Unless otherwise noted, any function that accepts valid indices can also be called with *pseudo-indices*, which represent some Lua values that are accessible to the C code but are not in the stack. Pseudo-indices are used to access the global environment, the registry, and the upvalues of a C function (see §3.17).

3.3 Stack Manipulation

The API offers the following functions for basic stack manipulation:

```
void lua_settop    (lua_State *L, int index);
void lua_pushvalue (lua_State *L, int index);
void lua_remove    (lua_State *L, int index);
void lua_insert    (lua_State *L, int index);
void lua_replace   (lua_State *L, int index);
```

`lua_settop` accepts any acceptable index, or 0, and sets the stack top to that 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. A useful macro defined in the `lua.h` is

```
#define lua_pop(L,n)   lua_settop(L, -(n)-1)
```

which pops `n` elements from the stack.

`lua_pushvalue` pushes onto the stack a copy of the element at the given index. `lua_remove` removes the element at the given position, shifting down the elements above that position to fill the gap. `lua_insert` moves the top element into the given position, shifting up the elements above that position to open space. `lua_replace` moves the top element into the given position, without shifting any element (therefore replacing the value at the given position). All these functions accept only valid indices. (You cannot call `lua_remove` or `lua_insert` with pseudo-indices, as they do not represent a stack position.)

As an example, if the stack starts as `10 20 30 40 50*` (from bottom to top; the '`*`' marks the top), then

```
lua_pushvalue(L, 3)    --> 10 20 30 40 50 30*
lua_pushvalue(L, -1)   --> 10 20 30 40 50 30 30*
lua_remove(L, -3)      --> 10 20 30 40 30 30*
lua_remove(L,  6)      --> 10 20 30 40 30*
lua_insert(L,  1)      --> 30 10 20 30 40*
lua_insert(L, -1)      --> 30 10 20 30 40*  (no effect)
lua_replace(L, 2)      --> 30 40 20 30*
lua_settop(L, -3)      --> 30 40*
lua_settop(L,  6)      --> 30 40 nil nil nil nil*
```

3.4 Querying the Stack

To check the type of a stack element, the following functions are available:

```
int lua_type            (lua_State *L, int index);
int lua_isnil           (lua_State *L, int index);
int lua_isboolean       (lua_State *L, int index);
int lua_isnumber        (lua_State *L, int index);
int lua_isstring        (lua_State *L, int index);
int lua_istable         (lua_State *L, int index);
int lua_isfunction      (lua_State *L, int index);
int lua_iscfunction     (lua_State *L, int index);
int lua_isuserdata      (lua_State *L, int index);
int lua_islightuserdata (lua_State *L, int index);
```

These functions can be called with any acceptable index.

`lua_type` returns the type of a value in the stack, or `LUA_TNONE` for a non-valid index (that is, if that stack position is "empty"). The types returned by `lua_type` are coded by the following constants defined in `lua.h`: `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD`, `LUA_TLIGHTUSERDATA`. The following function translates these constants to strings:

```
const char *lua_typename  (lua_State *L, int type);
```

The `lua_is*` functions return 1 if the object is compatible with the given type, and 0 otherwise. `lua_isboolean` is an exception to this rule: It succeeds only for boolean values (otherwise it would be useless, as any value has a boolean value). They always return 0 for a non-valid index. `lua_isnumber` accepts numbers and numerical strings; `lua_isstring` accepts strings and numbers (see §2.2.1); `lua_isfunction` accepts both Lua functions and C functions; and `lua_isuserdata` accepts both full and light userdata. To distinguish between Lua functions and C functions, you can use `lua_iscfunction`. To distinguish between full and light userdata, you can use `lua_islightuserdata`. To distinguish between numbers and numerical strings, you can use `lua_type`.

The API also contains functions to compare two values in the stack:

```
int lua_equal    (lua_State *L, int index1, int index2);
int lua_rawequal (lua_State *L, int index1, int index2);
int lua_lessthan (lua_State *L, int index1, int index2);
```

`lua_equal` and `lua_lessthan` are equivalent to their counterparts in Lua (see §2.5.2). `lua_rawequal` compares the values for primitive equality, without metamethods. These functions return 0 (false) if any of the indices are non-valid.

3.5 Getting Values from the Stack

To translate a value in the stack to a specific C type, you can use the following conversion functions:

```
int            lua_toboolean   (lua_State *L, int index);
lua_Number     lua_tonumber    (lua_State *L, int index);
const char    *lua_tostring    (lua_State *L, int index);
size_t         lua_strlen      (lua_State *L, int index);
lua_CFunction  lua_tocfunction (lua_State *L, int index);
void          *lua_touserdata  (lua_State *L, int index);
lua_State     *lua_tothread    (lua_State *L, int index);
void          *lua_topointer   (lua_State *L, int index);
```

These functions can be called with any acceptable index. When called with a non-valid index, they act as if the given value had an incorrect type.

`lua_toboolean` converts the Lua value at the given 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 real boolean values, use `lua_isboolean` to test the type of the value.)

`lua_tonumber` converts the Lua value at the given index to a number (by default, `lua_Number` is `double`). The Lua value must be a number or a string convertible to number (see §2.2.1); otherwise, `lua_tonumber` returns 0.

`lua_tostring` converts the Lua value at the given index to a string (`const char*`). The Lua value must be a string or a number; otherwise, the function returns `NULL`. If the value is a number, then `lua_tostring` also *changes the actual value in the stack to a string.* (This change confuses `lua_next` when `lua_tostring` is applied to keys.) `lua_tostring` returns a fully aligned pointer to a string inside the Lua state. This string always has a zero (`'\0'`) after its last character (as in C), but may contain other zeros in its body. If you do not know whether a string may contain zeros, you can use `lua_strlen` to get its actual length. Because Lua has garbage collection, there is no guarantee that the pointer returned by `lua_tostring` will be valid after the corresponding value is removed from the stack. If you need the string after the current function returns, then you should duplicate it or put it into the registry (see §3.18).

`lua_tocfunction` converts a value in the stack to a C function. This value must be a C function; otherwise, `lua_tocfunction` returns `NULL`. The type `lua_CFunction` is explained in §3.16.

`lua_tothread` converts a value in the stack to a Lua thread (represented as `lua_State *`). This value must be a thread; otherwise, `lua_tothread` returns `NULL`.

`lua_topointer` converts a value in the stack to a generic C pointer (`void *`). The value may be a userdata, a table, a thread, or a function; otherwise, `lua_topointer` returns `NULL`. Lua ensures that different objects of the same type return different pointers. There is no direct way to convert the pointer back to its original value. Typically this function is used for debug information.

`lua_touserdata` is explained in §3.8.

3.6 Pushing Values onto the Stack

The API has the following functions to push C values onto the stack:

```
void lua_pushboolean       (lua_State *L, int b);
void lua_pushnumber        (lua_State *L, lua_Number n);
void lua_pushlstring       (lua_State *L, const char *s, size_t len);
void lua_pushstring        (lua_State *L, const char *s);
void lua_pushnil           (lua_State *L);
void lua_pushcfunction     (lua_State *L, lua_CFunction f);
void lua_pushlightuserdata (lua_State *L, void *p);
```

These functions receive a C value, convert it to a corresponding Lua value, and push the result onto the stack. In particular, `lua_pushlstring` and `lua_pushstring` make an internal copy of the given string. `lua_pushstring` can only be used to push proper C strings (that is, strings that end with a zero and do not contain embedded zeros); otherwise, you should use the more general `lua_pushlstring`, which accepts an explicit size.

You can also push "formatted" strings:

```
const char *lua_pushfstring  (lua_State *L, const char *fmt, ...);
const char *lua_pushvfstring (lua_State *L, const char *fmt, va_list argp);
```

These functions push onto the stack a formatted string and return a pointer to that string. They are similar to `sprintf` and `vsprintf`, but with some important differences:

- You do not have to allocate the space for the result: The result is a Lua string and Lua takes care of memory allocation (and deallocation, through garbage collection).
- The conversion specifiers are quite restricted. There are no flags, widths, or precisions. The conversion specifiers can be simply '`%%`' (inserts a '`%`' in the string), '`%s`' (inserts a zero-terminated string, with no size restrictions), '`%f`' (inserts a `lua_Number`), '`%d`' (inserts an `int`), and '`%c`' (inserts an `int` as a character).

The function

```
void lua_concat (lua_State *L, int n);
```

concatenates the `n` values at the top of the stack, pops them, and leaves the result at the top. If `n` is 1, the result is that single string (that is, the function does nothing); if `n` is 0, the result is the empty string. Concatenation is done following the usual semantics of Lua (see §2.5.4).

3.7 Controlling Garbage Collection

Lua uses two numbers to control its garbage collection: the *count* and the *threshold* (see §2.9). The first counts the amount of memory in use by Lua; when the count reaches the threshold, Lua runs its garbage collector. After the collection, the count is updated and the threshold is set to twice the count value.

You can access the current values of these two numbers through the following functions:

```
int  lua_getgccount     (lua_State *L);
int  lua_getgcthreshold (lua_State *L);
```

Both return their respective values in Kbytes. You can change the threshold value with

```
void  lua_setgcthreshold (lua_State *L, int newthreshold);
```

Again, the `newthreshold` value is given in Kbytes. When you call this function, Lua sets the new threshold and checks it against the byte counter. If the new threshold is less than the byte counter, then Lua immediately runs the garbage collector. In particular `lua_setgcthreshold(L,0)` forces a garbage collection. After the collection, a new threshold is set according to the previous rule.

3.8 Userdata

Userdata represents C values in Lua. Lua supports two types of userdata: *full userdata* and *light userdata*.

A full userdata represents a block of memory. It is an object (like a table): You must create it, it can have its own metatable, and you can detect when it is being collected. A full userdata is only equal to itself (under raw equality).

A light userdata represents a pointer. It is a value (like a number): You do not create it, it has no metatables, it is not collected (as it was never created). A light userdata is equal to "any" light userdata with the same C address.

In Lua code, there is no way to test whether a userdata is full or light; both have type `userdata`. In C code, `lua_type` returns `LUA_TUSERDATA` for full userdata, and `LUA_TLIGHTUSERDATA` for light userdata.

You can create a new full userdata with the following function:

```
void *lua_newuserdata (lua_State *L, size_t size);
```

This function allocates a new block of memory with the given size, pushes on the stack a new userdata with the block address, and returns this address.

To push a light userdata into the stack you use `lua_pushlightuserdata` (see §3.6).

`lua_touserdata` (see §3.5) retrieves the value of a userdata. When applied on a full userdata, it returns the address of its block; when applied on a light userdata, it returns its pointer; when applied on a non-userdata value, it returns `NULL`.

When Lua collects a full userdata, it calls the userdata's `gc` metamethod, if any, and then it frees the userdata's corresponding memory.

3.9 Metatables

The following functions allow you to manipulate the metatables of an object:

```
int lua_getmetatable (lua_State *L, int index);
int lua_setmetatable (lua_State *L, int index);
```

`lua_getmetatable` pushes on the stack the metatable of a given object. If the index is not valid, or if the object does not have a metatable, `lua_getmetatable` returns 0 and pushes nothing on the stack.

`lua_setmetatable` pops a table from the stack and sets it as the new metatable for the given object. `lua_setmetatable` returns 0 when it cannot set the metatable of the given object (that is, when the object is neither a userdata nor a table); even then it pops the table from the stack.

3.10 Loading Lua Chunks

You can load a Lua chunk with `lua_load`:

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

int lua_load (lua_State *L, lua_Chunkreader reader, void *data,
                            const char *chunkname);
```

The return values of `lua_load` are:

- 0 — no errors;
- `LUA_ERRSYNTAX` — syntax error during pre-compilation.
- `LUA_ERRMEM` — memory allocation error.

If there are no errors, `lua_load` pushes the compiled chunk as a Lua function on top of the stack. Otherwise, it pushes an error message.

`lua_load` automatically detects whether the chunk is text or binary, and loads it accordingly (see program `luac`).

`lua_load` uses a user-supplied *reader* function to read the chunk. Every time it needs another piece of the chunk, `lua_load` calls the reader, passing along its `data` parameter. The reader must return a pointer to a block of memory with a new piece of the chunk and set `size` to the block size. To signal the end of the chunk, the reader returns `NULL`. The reader function may return pieces of any size greater than zero.

In the current implementation, the reader function cannot call any Lua function; to ensure that, it always receives `NULL` as the Lua state.

The *chunkname* is used for error messages and debug information (see §4).

See the auxiliary library (`lauxlib.c`) for examples of how to use `lua_load` and for some ready-to-use functions to load chunks from files and strings.

3.11 Manipulating Tables

Tables are created by calling the function

```
void lua_newtable (lua_State *L);
```

This function creates a new, empty table and pushes it onto the stack.

To read a value from a table that resides somewhere in the stack, call

```
void lua_gettable (lua_State *L, int index);
```

where `index` points to the table. `lua_gettable` pops a key from the stack and returns (on the stack) the contents of the table at that key. The table is left where it was in the stack. As in Lua, this function may trigger a metamethod for the "index" event (see §2.8). To get the real value of any table key, without invoking any metamethod, use the *raw* version:

```
void lua_rawget (lua_State *L, int index);
```

To store a value into a table that resides somewhere in the stack, you push the key and then the value onto the stack, and call

```
void lua_settable (lua_State *L, int index);
```

where `index` points to the table. `lua_settable` pops from the stack both the key and the value. The table is left where it was in the stack. As in Lua, this operation may trigger a metamethod for the "settable" or "newindex" events. To set the real value of any table index, without invoking any metamethod, use the *raw* version:

```
void lua_rawset (lua_State *L, int index);
```

You can traverse a table with the function

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

where `index` points to the table to be traversed. The function pops a key from the stack, and pushes a key-value pair from the table (the "next" pair after the given key). If there are no more elements, then `lua_next` returns 0 (and pushes nothing). Use a **nil** key to signal the start of a traversal.

A typical traversal looks like this:

```
/* table is in the stack at index 't' */
lua_pushnil(L);  /* first key */
while (lua_next(L, t) != 0) {
  /* 'key' is at index -2 and 'value' at index -1 */
  printf("%s - %s\n",
    lua_typename(L, lua_type(L, -2)), lua_typename(L, lua_type(L, -1)));
  lua_pop(L, 1);  /* removes 'value'; keeps 'key' for next iteration */
}
```

While traversing a table, do not call `lua_tostring` directly on a key, unless you know that the key is actually a string. Recall that `lua_tostring` *changes* the value at the given index; this confuses the next call to `lua_next`.

3.12 Manipulating Environments

All global variables are kept in ordinary Lua tables, called environments. The initial environment is called the global environment. This table is always at pseudo-index `LUA_GLOBALSINDEX`.

To access and change the value of global variables, you can use regular table operations over an environment table. For instance, to access the value of a global variable, do

```
lua_pushstring(L, varname);
lua_gettable(L, LUA_GLOBALSINDEX);
```

You can change the global environment of a Lua thread using `lua_replace`.

The following functions get and set the environment of Lua functions:

```
void lua_getfenv (lua_State *L, int index);
int  lua_setfenv (lua_State *L, int index);
```

`lua_getfenv` pushes on the stack the environment table of the function at index `index` in the stack. If the function is a C function, `lua_getfenv` pushes the global environment. `lua_setfenv` pops a table from the stack and sets it as the new environment for the function at index `index` in the stack. If the object at the given index is not a Lua function, `lua_setfenv` returns 0.

3.13 Using Tables as Arrays

The API has functions that help to use Lua tables as arrays, that is, tables indexed by numbers only:

```
void lua_rawgeti (lua_State *L, int index, int n);
void lua_rawseti (lua_State *L, int index, int n);
```

`lua_rawgeti` pushes the value of the n-th element of the table at stack position `index`. `lua_rawseti` sets the value of the n-th element of the table at stack position `index` to the value at the top of the stack, removing this value from the stack.

3.14 Calling Functions

Functions defined in Lua and C functions registered in Lua can be called from the host program. This is done using the following protocol: First, the function to be called is pushed onto the stack; then, the arguments to the function are pushed in *direct order*, that is, the first argument is pushed first. Finally, the function is called using

```
void lua_call (lua_State *L, int nargs, int nresults);
```

`nargs` is the number of arguments that you pushed onto the stack. All arguments and the function value are popped from the stack, and the function results are pushed. The number of results are adjusted to `nresults`, unless `nresults` is `LUA_MULTRET`. In that case, *all* results from the function are pushed. Lua takes care that the returned values fit into the stack space. The function results are pushed onto the stack in direct order (the first result is pushed first), so that after the call the last result is on the top.

The following example shows how the host program may do the equivalent to this Lua code:

```
a = f("how", t.x, 14)
```

Here it is in C:

```
lua_pushstring(L, "t");
lua_gettable(L, LUA_GLOBALSINDEX);          /* global 't' (for later use) */
lua_pushstring(L, "a");                                  /* var name */
lua_pushstring(L, "f");                               /* function name */
lua_gettable(L, LUA_GLOBALSINDEX);              /* function to be called */
lua_pushstring(L, "how");                                /* 1st argument */
lua_pushstring(L, "x");                            /* push the string "x" */
lua_gettable(L, -5);                       /* push result of t.x (2nd arg) */
lua_pushnumber(L, 14);                                   /* 3rd argument */
lua_call(L, 3, 1);         /* call function with 3 arguments and 1 result */
lua_settable(L, LUA_GLOBALSINDEX);             /* set global variable 'a' */
lua_pop(L, 1);                                /* remove 't' from the stack */
```

Note that the code above is "balanced": at its end, the stack is back to its original configuration. This is considered good programming practice.

(We did this example using only the raw functions provided by Lua's API, to show all the details. Usually programmers define and use several macros and auxiliary functions that provide higher level access to Lua. See the source code of the standard libraries for examples.)

3.15 Protected Calls

When you call a function with `lua_call`, any error inside the called function is propagated upwards (with a `longjmp`). If you need to handle errors, then you should use `lua_pcall`:

```
int lua_pcall (lua_State *L, int nargs, int nresults, int errfunc);
```

Both `nargs` and `nresults` have the same meaning as in `lua_call`. If there are no errors during the call, `lua_pcall` behaves exactly like `lua_call`. However, if there is any error, `lua_pcall` catches it, pushes a single value at the stack (the error message), and returns an error code. Like `lua_call`, `lua_pcall` always removes the function and its arguments from the stack.

If `errfunc` is 0, then the error message returned is exactly the original error message. Otherwise, `errfunc` gives the stack index for an *error handler function*. (In the current implementation, that index cannot be a pseudo-index.) In case of runtime errors, that function will be called with the error message and its return value will be the message returned by `lua_pcall`.

Typically, the error handler function is used to add more debug information to the error message, such as a stack traceback. Such information cannot be gathered after the return of `lua_pcall`, since by then the stack has unwound.

The `lua_pcall` function returns 0 in case of success or one of the following error codes (defined in `lua.h`):

- `LUA_ERRRUN` — a runtime error.
- `LUA_ERRMEM` — memory allocation error. For such errors, Lua does not call the error handler function.
- `LUA_ERRERR` — error while running the error handler function.

3.16 Defining C Functions

Lua can be extended with functions written in C. These functions must be of type `lua_CFunction`, which is defined as

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

A C function receives a Lua state and returns an integer, the number of values it wants to return to Lua.

In order to communicate properly with Lua, a C function must follow the following protocol, which defines the way parameters and results are passed: A C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first). So, when the function starts, its first argument (if any) is at index 1. To return values to Lua, a C function just pushes them onto the stack, in direct order (the first result is pushed first), and returns the number of results. Any other value in the stack below the results will be properly discharged by Lua. Like a Lua function, a C function called by Lua can also return many results.

As an example, the following function receives a variable number of numerical arguments and returns their average and sum:

```
static int foo (lua_State *L) {
  int n = lua_gettop(L);    /* number of arguments */
  lua_Number sum = 0;
  int i;
  for (i = 1; i <= n; i++) {
    if (!lua_isnumber(L, i)) {
      lua_pushstring(L, "incorrect argument to function ‘average’");
      lua_error(L);
    }
    sum += lua_tonumber(L, i);
  }
  lua_pushnumber(L, sum/n);        /* first result */
  lua_pushnumber(L, sum);         /* second result */
  return 2;                   /* number of results */
}
```

To register a C function to Lua, there is the following convenience macro:

```
#define lua_register(L,n,f) \
        (lua_pushstring(L, n), \
         lua_pushcfunction(L, f), \
         lua_settable(L, LUA_GLOBALSINDEX))
/* lua_State *L;    */
/* const char *n;   */
/* lua_CFunction f; */
```

which receives the name the function will have in Lua and a pointer to the function. Thus, the C function `foo` above may be registered in Lua as `average` by calling

```
lua_register(L, "average", foo);
```

3.17 Defining C Closures

When a C function is created, it is possible to associate some values with it, thus creating a *C closure*; these values are then accessible to the function whenever it is called. To associate values with a C function, first these values should be pushed onto the stack (when there are multiple values, the first value is pushed first). Then the function

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn, int n);
```

is used to push the C function onto the stack, with the argument `n` telling how many values should be associated with the function (`lua_pushcclosure` also pops these values from the stack); in fact, the macro `lua_pushcfunction` is defined as `lua_pushcclosure` with `n` set to 0.

Then, whenever the C function is called, those values are located at specific pseudo-indices. Those pseudo-indices are produced by a macro `lua_upvalueindex`. The first value associated with a function is at position `lua_upvalueindex(1)`, and so on. Any access to `lua_upvalueindex(`n`)`, where n is greater than the number of upvalues of the current function, produces an acceptable (but invalid) index.

For examples of C functions and closures, see the standard libraries in the official Lua distribution (`src/lib/*.c`).

3.18 Registry

Lua provides a registry, a pre-defined table that can be used by any C code to store whatever Lua value it needs to store, specially if the C code needs to keep that Lua value outside the life span of a C function. This table is always located at pseudo-index `LUA_REGISTRYINDEX`. Any C library can store data into this table, as long as it chooses keys different from other libraries. Typically, you should use as key a string containing your library name or a light userdata with the address of a C object in your code.

The integer keys in the registry are used by the reference mechanism, implemented by the auxiliary library, and therefore should not be used by other purposes.

3.19 Error Handling in C

Internally, Lua uses the C `longjmp` facility to handle errors. When Lua faces any error (such as memory allocation errors, type errors, syntax errors) it *raises* an error, that is, it does a long jump. A *protected environment* uses `setjmp` to set a recover point; any error jumps to the most recent active recover point.

If an error happens outside any protected environment, Lua calls a *panic function* and then calls `exit(EXIT_FAILURE)`. You can change the panic function with

```
lua_CFunction lua_atpanic (lua_State *L, lua_CFunction panicf);
```

Your new panic function may avoid the application exit by never returning (e.g., by doing a long jump). Nevertheless, the corresponding Lua state will not be consistent; the only safe operation with it is to close it.

Almost any function in the API may raise an error, for instance due to a memory allocation error. The following functions run in protected mode (that is, they create a protected environment to run), so they never raise an error: `lua_open`, `lua_close`, `lua_load`, and `lua_pcall`.

There is yet another function that runs a given C function in protected mode:

```
int lua_cpcall (lua_State *L, lua_CFunction func, void *ud);
```

`lua_cpcall` calls `func` in protected mode. `func` starts with only one element in its stack, a light userdata containing `ud`. In case of errors, `lua_cpcall` returns the same error codes as `lua_pcall` (see §3.15), plus the error object on the top of the stack; otherwise, it returns zero, and does not change the stack. Any value returned by `func` is discarded.

C code can generate a Lua error calling the function

```
void lua_error (lua_State *L);
```

The error message (which actually can be any type of object) must be on the stack top. This function does a long jump, and therefore never returns.

3.20 Threads

Lua offers partial support for multiple threads of execution. If you have a C library that offers multi-threading, then Lua can cooperate with it to implement the equivalent facility in Lua. Also, Lua implements its own coroutine system on top of threads. The following function creates a new thread in Lua:

```
lua_State *lua_newthread (lua_State *L);
```

This function pushes the thread on the stack and returns a pointer to a `lua_State` that represents this new thread. The new state returned by this function shares with the original state all global objects (such as tables), but has an independent run-time stack.

Each thread has an independent global environment table. When you create a thread, this table is the same as that of the given state, but you can change each one independently.

There is no explicit function to close or to destroy a thread. Threads are subject to garbage collection, like any Lua object.

To manipulate threads as coroutines, Lua offers the following functions:

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

To start a coroutine, you first create a new thread; then you push on its stack the body function plus any eventual arguments; then you call `lua_resume`, with `narg` being the number of arguments. This call returns when the coroutine suspends or finishes its execution. When it returns, the stack contains all values passed to `lua_yield`, or all values returned by the body function. `lua_resume` returns 0 if there are no errors running the coroutine, or an error code (see §3.15). In case of errors, the stack contains only the error message. To restart a coroutine, you put on its stack only the values to be passed as results from `yield`, and then call `lua_resume`.

The `lua_yield` function can only be called as the return expression of a C function, as follows:

```
return lua_yield (L, nresults);
```

When a C function calls `lua_yield` in that way, the running coroutine suspends its execution, and the call to `lua_resume` that started this coroutine returns. The parameter `nresults` is the number of values from the stack that are passed as results to `lua_resume`.

To exchange values between different threads, you may use `lua_xmove`:

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

It pops `n` values from the stack `from`, and puhses them into the stack `to`.

4 The Debug Interface

Lua has no built-in debugging facilities. Instead, it offers a special interface by means of functions and *hooks*. This interface allows the construction of different kinds of debuggers, profilers, and other tools that need "inside information" from the interpreter.

4.1 Stack and Function Information

The main function to get information about the interpreter runtime stack is

```
int lua_getstack (lua_State *L, int level, lua_Debug *ar);
```

This function fills parts of a `lua_Debug` structure with an identification of the *activation record* of the function executing at a given level. Level 0 is the current running function, whereas level $n+1$ is the function that has called level n. When there are no errors, `lua_getstack` returns 1; when called with a level greater than the stack depth, it returns 0.

The structure `lua_Debug` is used to carry different pieces of information about an active function:

```
typedef struct lua_Debug {
  int event;
  const char *name;      /* (n) */
  const char *namewhat;  /* (n) 'global', 'local', 'field', 'method' */
  const char *what;      /* (S) 'Lua' function, 'C' function, Lua 'main' */
  const char *source;    /* (S) */
  int currentline;       /* (l) */
  int nups;              /* (u) number of upvalues */
  int linedefined;       /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */

  /* private part */
  ...
} lua_Debug;
```

`lua_getstack` fills only the private part of this structure, for later use. To fill the other fields of `lua_Debug` with useful information, call

```
int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
```

This function returns 0 on error (for instance, an invalid option in `what`). Each character in the string `what` selects some fields of the structure `ar` to be filled, as indicated by the letter in parentheses in the definition of `lua_Debug` above: '`S`' fills in the fields `source`, `linedefined`, and `what`; '`l`' fills in the field `currentline`, etc. Moreover, '`f`' pushes onto the stack the function that is running at the given level.

To get information about a function that is not active (that is, not in the stack), you push it onto the stack and start the `what` string with the character '`>`'. For instance, to know in which line a function `f` was defined, you can write

```
lua_Debug ar;
lua_pushstring(L, "f");
lua_gettable(L, LUA_GLOBALSINDEX);  /* get global 'f' */
lua_getinfo(L, ">S", &ar);
printf("%d\n", ar.linedefined);
```

The fields of `lua_Debug` have the following meaning:

`source` If the function was defined in a string, then `source` is that string. If the function was defined in a file, then `source` starts with a '`@`' followed by the file name.

`short_src` A "printable" version of `source`, to be used in error messages.

`linedefined` the line number where the definition of the function starts.

`what` the string `"Lua"` if this is a Lua function, `"C"` if this is a C function, `"main"` if this is the main part of a chunk, and `"tail"` if this was a function that did a tail call. In the latter case, Lua has no other information about this function.

`currentline` the current line where the given function is executing. When no line information is available, `currentline` is set to -1.

`name` a reasonable name for the given function. Because functions in Lua are first class values, they do not have a fixed name: Some functions may be the value of multiple global variables, while others may be stored only in a table field. The `lua_getinfo` function checks how the function was called or whether it is the value of a global variable to find a suitable name. If it cannot find a name, then `name` is set to `NULL`.

`namewhat` Explains the `name` field. The value of `namewhat` can be `"global"`, `"local"`, `"method"`, `"field"`, or `""` (the empty string), according to how the function was called. (Lua uses the empty string when no other option seems to apply.)

`nups` The number of upvalues of the function.

4.2 Manipulating Local Variables and Upvalues

For the manipulation of local variables and upvalues, the debug interface uses indices: The first parameter or local variable has index 1, and so on, until the last active local variable. Upvalues have no particular order, as they are active through the whole function.

The following functions allow the manipulation of the local variables of a given activation record:

```
const char *lua_getlocal (lua_State *L, const lua_Debug *ar, int n);
const char *lua_setlocal (lua_State *L, const lua_Debug *ar, int n);
```

The parameter `ar` must be a valid activation record that was filled by a previous call to `lua_getstack` or given as argument to a hook (see §4.3). `lua_getlocal` gets the index `n` of a local variable, pushes the variable's value onto the stack, and returns its name. `lua_setlocal` assigns the value at the top of the stack to the variable and returns its name. Both functions return `NULL` when the index is greater than the number of active local variables.

The following functions allow the manipulation of the upvalues of a given function (unlike local variables, the upvalues of a function are accessible even when the function is not active):

```
const char *lua_getupvalue (lua_State *L, int funcindex, int n);
const char *lua_setupvalue (lua_State *L, int funcindex, int n);
```

These functions operate both on Lua functions and on C functions. (For Lua functions, upvalues are the external local variables that the function uses, and that consequently are included in its closure.) `funcindex` points to a function in the stack. `lua_getpuvalue` gets the index `n` of an upvalue, pushes the upvalue's value onto the stack, and returns its name. `lua_setupvalue` assigns the value at the top of the stack to the upvalue and returns its name. Both functions return `NULL` when the index is greater than the number of upvalues. For C functions, these functions use the empty string `""` as a name for all upvalues.

As an example, the following function lists the names of all local variables and upvalues for a function at a given level of the stack:

```
int listvars (lua_State *L, int level) {
  lua_Debug ar;
  int i;
  const char *name;
  if (lua_getstack(L, level, &ar) == 0)
    return 0;  /* failure: no such level in the stack */
  i = 1;
  while ((name = lua_getlocal(L, &ar, i++)) != NULL) {
    printf("local %d %s\n", i-1, name);
    lua_pop(L, 1);  /* remove variable value */
  }
  lua_getinfo(L, "f", &ar);  /* retrieves function */
  i = 1;
  while ((name = lua_getpuvalue(L, -1, i++)) != NULL) {
    printf("upvalue %d %s\n", i-1, name);
    lua_pop(L, 1);  /* remove upvalue value */
  }
  return 1;
}
```

4.3 Hooks

Lua offers a mechanism of hooks, which are user-defined C functions that are called during the program execution. A hook may be called in four different events: a *call* event, when Lua calls a function; a *return* event, when Lua returns from a function; a *line* event, when Lua starts executing a new line of code; and a *count* event, which happens every "count" instructions. Lua identifies these events with the following constants: `LUA_HOOKCALL`, `LUA_HOOKRET` (or `LUA_HOOKTAILRET`, see below), `LUA_HOOKLINE`, and `LUA_HOOKCOUNT`.

A hook has type `lua_Hook`, defined as follows:

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
```

You can set the hook with the following function:

```
int lua_sethook (lua_State *L, lua_Hook func, int mask, int count);
```

`func` is the hook. `mask` specifies on which events the hook will be called: It is formed by a disjunction of the constants `LUA_MASKCALL`, `LUA_MASKRET`, `LUA_MASKLINE`, and `LUA_MASKCOUNT`. The `count` argument is only meaningful when the mask includes `LUA_MASKCOUNT`. For each event, the hook is called as explained below:

The call hook is called when the interpreter calls a function. The hook is called just after Lua enters the new function.

The return hook is called when the interpreter returns from a function. The hook is called just before Lua leaves the function.

The line hook is called when the interpreter is about to start the execution of a new line of code, or when it jumps back in the code (even to the same line). (This event only happens while Lua is executing a Lua function.)

The count hook is called after the interpreter executes every `count` instructions. (This event only happens while Lua is executing a Lua function.)

A hook is disabled by setting `mask` to zero.

You can get the current hook, the current mask, and the current count with the next functions:

```
lua_Hook lua_gethook      (lua_State *L);
int      lua_gethookmask  (lua_State *L);
int      lua_gethookcount (lua_State *L);
```

Whenever a hook is called, its `ar` argument has its field `event` set to the specific event that triggered the hook. Moreover, for line events, the field `currentline` is also set. To get the value of any other field in `ar`, the hook must call `lua_getinfo`. For return events, `event` may be `LUA_HOOKRET`, the normal value, or `LUA_HOOKTAILRET`. In the latter case, Lua is simulating a return from a function that did a tail call; in this case, it is useless to call `lua_getinfo`.

While Lua is running a hook, it disables other calls to hooks. Therefore, if a hook calls back Lua to execute a function or a chunk, that execution occurs without any calls to hooks.

5 Standard Libraries

The standard libraries provide useful functions that are implemented directly through the C API. Some of these functions provide essential services to the language (e.g., `type` and `getmetatable`); others provide access to "outside" services (e.g., I/O); and others could be implemented in Lua itself, but are quite useful or have critical performance to deserve an implementation in C (e.g., `sort`).

All libraries are implemented through the official C API and are provided as separate C modules. Currently, Lua has the following standard libraries:

- basic library;
- string manipulation;
- table manipulation;
- mathematical functions (sin, log, etc.);
- input and output;
- operating system facilities;
- debug facilities.

Except for the basic library, each library provides all its functions as fields of a global table or as methods of its objects.

To have access to these libraries, the C host program must first call the functions `luaopen_base` (for the basic library), `luaopen_string` (for the string library), `luaopen_table` (for the table library), `luaopen_math` (for the mathematical library), `luaopen_io` (for the I/O and the Operating System libraries), and `luaopen_debug` (for the debug library). These functions are declared in `lualib.h`.

5.1 Basic Functions

The basic library provides some core functions to Lua. If you do not include this library in your application, you should check carefully whether you need to provide some alternative implementation for some of its facilities.

• `assert (v [, message])`

Issues an error when the value of its argument `v` is **nil** or **false**; otherwise, returns this value. `message` is an error message; when absent, it defaults to "assertion failed!"

• `collectgarbage ([limit])`

Sets the garbage-collection threshold to the given limit (in Kbytes) and checks it against the byte counter. If the new threshold is smaller than the byte counter, then Lua immediately runs the garbage collector (see §2.9). If `limit` is absent, it defaults to zero (thus forcing a garbage-collection cycle).

• `dofile (filename)`

Opens the named file and executes its contents as a Lua chunk. When called without arguments, `dofile` executes the contents of the standard input (`stdin`). Returns any value returned by the chunk. In case of errors, `dofile` propagates the error to its caller (that is, it does not run in protected mode).

• `error (message [, level])`

Terminates the last protected function called and returns `message` as the error message. Function `error` never returns.

The `level` argument specifies where the error message points the error. With level 1 (the default), the error position is where the `error` function was called. Level 2 points the error to where the function that called `error` was called; and so on.

• `_G`

A global variable (not a function) that holds the global environment (that is, `_G._G = _G`). Lua itself does not use this variable; changing its value does not affect any environment. (Use `setfenv` to change environments.)

• `getfenv (f)`

Returns the current environment in use by the function. `f` can be a Lua function or a number, which specifies the function at that stack level: Level 1 is the function calling `getfenv`. If the given function is not a Lua function, or if `f` is 0, `getfenv` returns the global environment. The default for `f` is 1.

If the environment has a `"__fenv"` field, returns the associated value, instead of the environment.

• `getmetatable (object)`

If the object does not have a metatable, returns **nil**. Otherwise, if the object's metatable has a `"__metatable"` field, returns the associated value. Otherwise, returns the metatable of the given object.

• `gcinfo ()`

Returns two results: the number of Kbytes of dynamic memory that Lua is using and the current garbage collector threshold (also in Kbytes).

• `ipairs (t)`

Returns an iterator function, the table `t`, and 0, so that the construction

```
for i,v in ipairs(t) do ... end
```

will iterate over the pairs (`1,t[1]`), (`2,t[2]`), ..., up to the first integer key with a nil value in the table.

• `loadfile (filename)`

Loads a file as a Lua chunk (without running it). If there are no errors, returns the compiled chunk as a function; otherwise, returns **nil** plus the error message. The environment of the returned function is the global environment.

• `loadlib (libname, funcname)`

Links the program with the dynamic C library `libname`. Inside this library, looks for a function `funcname` and returns this function as a C function.

`libname` must be the complete file name of the C library, including any eventual path and extension.

This function is not supported by ANSI C. As such, it is only available on some platforms (Windows, Linux, Solaris, BSD, plus other Unix systems that support the `dlfcn` standard).

• `loadstring (string [, chunkname])`

Loads a string as a Lua chunk (without running it). If there are no errors, returns the compiled chunk as a function; otherwise, returns **nil** plus the error message. The environment of the returned function is the global environment.

The optional parameter `chunkname` is the name to be used in error messages and debug information.

To load and run a given string, use the idiom

```
assert(loadstring(s))()
```

• `next (table [, index])`

Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. `next` returns the next index of the table and the value associated with the index. When called with **nil** as its second argument, `next` returns the first index of the table and its associated value. When called with the last index, or with **nil** in an empty table, `next` returns **nil**. If the second argument is absent, then it is interpreted as **nil**.

Lua has no declaration of fields; There is no difference between a field not present in a table or a field with value **nil**. Therefore, `next` only considers fields with non-**nil** values. The order in which the indices are enumerated is not specified, *even for numeric indices*. (To traverse a table in numeric order, use a numerical **for** or the `ipairs` function.)

The behavior of `next` is *undefined* if, during the traversal, you assign any value to a non-existent field in the table.

• `pairs (t)`

Returns the `next` function and the table `t` (plus a **nil**), so that the construction

```
for k,v in pairs(t) do ... end
```

will iterate over all key–value pairs of table `t`.

• `pcall (f, arg1, arg2, ...)`

Calls function `f` with the given arguments in protected mode. That means that any error inside `f` is not propagated; instead, `pcall` catches the error and returns a status code. Its first result is the status code (a boolean), which is **true** if the call succeeds without errors. In such case, `pcall` also returns all results from the call, after this first result. In case of any error, `pcall` returns **false** plus the error message.

• `print (e1, e2, ...)`

Receives any number of arguments, and prints their values in `stdout`, using the `tostring` function to convert them to strings. This function is not intended for formatted output, but only as a quick way to show a value, typically for debugging. For formatted output, use `format` (see §5.3).

• `rawequal (v1, v2)`

Checks whether `v1` is equal to `v2`, without invoking any metamethod. Returns a boolean.

• `rawget (table, index)`

Gets the real value of `table[index]`, without invoking any metamethod. `table` must be a table; `index` is any value different from **nil**.

• `rawset (table, index, value)`

Sets the real value of `table[index]` to `value`, without invoking any metamethod. `table` must be a table, `index` is any value different from **nil**, and `value` is any Lua value.

• `require (packagename)`

Loads the given package. The function starts by looking into the table `_LOADED` to determine whether `packagename` is already loaded. If it is, then `require` returns the value that the package returned when it was first loaded. Otherwise, it searches a path looking for a file to load.

If the global variable `LUA_PATH` is a string, this string is the path. Otherwise, `require` tries the environment variable `LUA_PATH`. As a last resort, it uses the predefined path `"?;?.lua"`.

The path is a sequence of *templates* separated by semicolons. For each template, `require` will change each interrogation mark in the template to `packagename`, and then will try to load the resulting file name. So, for instance, if the path is

```
"./?.lua;./?.lc;/usr/local/?/?.lua;/lasttry"
```

a `require "mod"` will try to load the files `./mod.lua`, `./mod.lc`, `/usr/local/mod/mod.lua`, and `/lasttry`, in that order.

The function stops the search as soon as it can load a file, and then it runs the file. After that, it associates, in table `_LOADED`, the package name with the value that the package returned, and returns that value. If the package returns **nil** (or no value), `require` converts this value to **true**. If the package returns **false**, `require` also returns **false**. However, as the mark in table `_LOADED` is **false**, any new attempt to reload the file will happen as if the package was not loaded (that is, the package will be loaded again).

If there is any error loading or running the file, or if it cannot find any file in the path, then `require` signals an error.

While running a file, `require` defines the global variable `_REQUIREDNAME` with the package name. The package being loaded always runs within the global environment.

• `setfenv (f, table)`

Sets the current environment to be used by the given function. `f` can be a Lua function or a number, which specifies the function at that stack level: Level 1 is the function calling `setfenv`.

As a special case, when `f` is 0 `setfenv` changes the global environment of the running thread.

If the original environment has a `"__fenv"` field, `setfenv` raises an error.

• `setmetatable (table, metatable)`

Sets the metatable for the given table. (You cannot change the metatable of a userdata from Lua.) If `metatable` is **nil**, removes the metatable of the given table. If the original metatable has a `"__metatable"` field, raises an error.

• `tonumber (e [, base])`

Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then `tonumber` returns that number; otherwise, it returns **nil**.

An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter '`A`' (in either upper or lower case) represents 10, '`B`' represents 11, and so forth, with '`Z`' representing 35. In base 10 (the default), the number may have a decimal part, as well as an optional exponent part (see §2.2.1). In other bases, only unsigned integers are accepted.

• `tostring (e)`

Receives an argument of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use `format` (see §5.3).

If the metatable of `e` has a `"__tostring"` field, `tostring` calls the corresponding value with `e` as argument, and uses the result of the call as its result.

• `type (v)`

Returns the type of its only argument, coded as a string. The possible results of this function are `"nil"` (a string, not the value **nil**), `"number"`, `"string"`, `"boolean`, `"table"`, `"function"`, `"thread"`, and `"userdata"`.

• `unpack (list)`

Returns all elements from the given list. This function is equivalent to

```
return list[1], list[2], ..., list[n]
```

except that the above code can be written only for a fixed n. The number n is the size of the list, as defined for the `table.getn` function.

• `_VERSION`

A global variable (not a function) that holds a string containing the current interpreter version. The current content of this string is `"Lua 5.0"`.

• `xpcall (f, err)`

This function is similar to `pcall`, except that you can set a new error handler.

`xpcall` calls function `f` in protected mode, using `err` as the error handler. Any error inside `f` is not propagated; instead, `xpcall` catches the error, calls the `err` function with the original error object, and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, `xpcall` also returns all results from the call, after this first result. In case of any error, `xpcall` returns false plus the result from `err`.

5.2 Coroutine Manipulation

The operations related to coroutines comprise a sub-library of the basic library and come inside the table `coroutine`. See §2.10 for a general description of coroutines.

• `coroutine.create (f)`

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns this new coroutine, an object with type `"thread"`.

• `coroutine.resume (co, val1, ...)`

Starts or continues the execution of coroutine `co`. The first time you resume a coroutine, it starts running its body. The arguments `val1`, ... go as the arguments to the body function. If the coroutine has yielded, `resume` restarts it; the arguments `val1`, ... go as the results from the yield.

If the coroutine runs without any errors, `resume` returns **true** plus any values passed to `yield` (if the coroutine yields) or any values returned by the body function (if the coroutine terminates). If there is any error, `resume` returns **false** plus the error message.

• `coroutine.status (co)`

Returns the status of coroutine `co`, as a string: `"running"`, if the coroutine is running (that is, it called `status`); `"suspended"`, if the coroutine is suspended in a call to `yield`, or if it has not started running yet; and `"dead"` if the coroutine has finished its body function, or if it has stopped with an error.

• `coroutine.wrap (f)`

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns a function that resumes the coroutine each time it is called. Any arguments passed to the function behave as the extra arguments to `resume`. Returns the same values returned by `resume`, except the first boolean. In case of error, propagates the error.

• `coroutine.yield (val1, ...)`

Suspends the execution of the calling coroutine. The coroutine cannot be running neither a C function, nor a metamethod, nor an iterator. Any arguments to `yield` go as extra results to `resume`.

5.3 String Manipulation

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Lua, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position −1, and so on.

The string library provides all its functions inside the table `string`.

• `string.byte (s [, i])`

Returns the internal numerical code of the `i`-th character of `s`, or **nil** if the index is out of range. If `i` is absent, then it is assumed to be 1. `i` may be negative.

Note that numerical codes are not necessarily portable across platforms.

• `string.char (i1, i2, ...)`

Receives 0 or more integers. Returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its correspondent argument.

Note that numerical codes are not necessarily portable across platforms.

• `string.dump (function)`

Returns a binary representation of the given function, so that a later `loadstring` on that string returns a copy of the function. `function` must be a Lua function without upvalues.

• `string.find (s, pattern [, init [, plain]])`

Looks for the first *match* of `pattern` in the string `s`. If it finds one, then `find` returns the indices of `s` where this occurrence starts and ends; otherwise, it returns **nil**. If the pattern specifies captures (see `string.gsub` below), the captured strings are returned as extra results. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and may be negative. A value of **true** as a fourth, optional argument `plain` turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in `pattern` being considered "magic". Note that if `plain` is given, then `init` must be given too.

• `string.len (s)`

Receives a string and returns its length. The empty string `""` has length 0. Embedded zeros are counted, so `"a\000b\000c"` has length 5.

• `string.lower (s)`

Receives a string and returns a copy of that string with all uppercase letters changed to lowercase. All other characters are left unchanged. The definition of what is an uppercase letter depends on the current locale.

• `string.rep (s, n)`

Returns a string that is the concatenation of `n` copies of the string `s`.

• `string.sub (s, i [, j])`

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` may be negative. If `j` is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call `string.sub(s,1,j)` returns a prefix of `s` with length `j`, and `string.sub(s, -i)` returns a suffix of `s` with length `i`.

• `string.upper (s)`

Receives a string and returns a copy of that string with all lowercase letters changed to uppercase. All other characters are left unchanged. The definition of what is a lowercase letter depends on the current locale.

• `string.format (formatstring, e1, e2, ...)`

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the `printf` family of standard C functions. The only differences are that the options/modifiers `*`, `l`, `L`, `n`, `p`, and `h` are not supported, and there is an extra option, `q`. The `q` option formats a string in a form suitable to be safely read back by the Lua interpreter: The string is written between double quotes,

and all double quotes, newlines, and backslashes in the string are correctly escaped when written. For instance, the call

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

will produce the string:

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

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string. The `*` modifier can be simulated by building the appropriate format string. For example, `"%*g"` can be simulated with `"%"..width.."g"`.

String values to be formatted with `%s` cannot contain embedded zeros.

• `string.gfind (s, pat)`

Returns an iterator function that, each time it is called, returns the next captures from pattern `pat` over string `s`.

If `pat` specifies no captures, then the whole match is produced in each call.

As an example, the following loop

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

will iterate over all the words from string `s`, printing one per line. The next example collects all pairs `key=value` from the given string into a table:

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

• `string.gsub (s, pat, repl [, n])`

Returns a copy of `s` in which all occurrences of the pattern `pat` have been replaced by a replacement string specified by `repl`. `gsub` also returns, as a second value, the total number of substitutions made.

If `repl` is a string, then its value is used for replacement. Any sequence in `repl` of the form `%`n, with n between 1 and 9, stands for the value of the n-th captured substring (see below).

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order; if the pattern specifies no captures, then the whole match is passed as a sole argument. If the value returned by this function is a string, then it is used as the replacement string; otherwise, the replacement string is the empty string.

The optional last parameter `n` limits the maximum number of substitutions to occur. For instance, when `n` is 1 only the first occurrence of `pat` is replaced.

Here are some examples:

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

x = string.gsub("hello world", "(%w+)", "%1 %1", 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="5.0"}
x = string.gsub("$name_$version.tar.gz", "%$(%w+)", function (v)
      return t[v]
    end)
--> x="lua_5.0.tar.gz"
```

Patterns

Character Class: A *character class* is used to represent a set of characters. The following combinations are allowed in describing a character class:

- ***x*** (where x is not one of the magic characters `^$()%.[]*+-?`) — represents the character x itself.
- `.` — (a dot) represents all characters.
- `%a` — represents all letters.
- `%c` — represents all control characters.
- `%d` — represents all digits.
- `%l` — represents all lowercase letters.
- `%p` — represents all punctuation characters.
- `%s` — represents all space characters.
- `%u` — represents all uppercase letters.
- `%w` — represents all alphanumeric characters.
- `%x` — represents all hexadecimal digits.
- `%z` — represents the character with representation 0.
- `%`*x* (where x is any non-alphanumeric character) — represents the character x. This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a '`%`' when used to represent itself in a pattern.

`[`*set*`]` — represents the class which is the union of all characters in *set*. A range of characters may be specified by separating the end characters of the range with a '`-`'. All classes `%`*x* described above may also be used as components in *set*. All other characters in *set* represent themselves. For example, `[%w_]` (or `[_%w]`) represents all alphanumeric characters plus the underscore, `[0-7]` represents the octal digits, and `[0-7%l%-]` represents the octal digits plus the lowercase letters plus the '`-`' character.

The interaction between ranges and classes is not defined. Therefore, patterns like `[%a-z]` or `[a-%%]` have no meaning.

`[^`*set*`]` — represents the complement of *set*, where *set* is interpreted as above.

For all classes represented by single letters (`%a`, `%c`, etc.), the corresponding uppercase letter represents the complement of the class. For instance, `%S` represents all non-space characters.

The definitions of letter, space, and other character groups depend on the current locale. In particular, the class `[a-z]` may not be equivalent to `%l`. The second form should be preferred for portability.

Pattern Item: A *pattern item* may be

- a single character class, which matches any single character in the class;
- a single character class followed by '`*`', which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '`+`', which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by '`-`', which also matches 0 or more repetitions of characters in the class. Unlike '`*`', these repetition items will always match the *shortest* possible sequence;
- a single character class followed by '`?`', which matches 0 or 1 occurrence of a character in the class;
- `%`*n*, for *n* between 1 and 9; such item matches a substring equal to the *n*-th captured string (see below);
- `%b`*xy*, where *x* and *y* are two distinct characters; such item matches strings that start with *x*, end with *y*, and where the *x* and *y* are *balanced*. This means that, if one reads the string from left to right, counting +1 for an *x* and −1 for a *y*, the ending *y* is the first *y* where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.

Pattern: A *pattern* is a sequence of pattern items. A '`^`' at the beginning of a pattern anchors the match at the beginning of the subject string. A '`$`' at the end of a pattern anchors the match at the end of the subject string. At other positions, '`^`' and '`$`' have no special meaning and represent themselves.

Captures: A pattern may contain sub-patterns enclosed in parentheses; they describe *captures*. When a match succeeds, the substrings of the subject string that match captures are stored (*captured*) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern `"(a*(.)%w(%s*))"`, the part of the string matching `"a*(.)%w(%s*)"` is stored as the first capture (and therefore has number 1); the character matching `.` is captured with number 2, and the part matching `%s*` has number 3.

As a special case, the empty capture `()` captures the current string position (a number). For instance, if we apply the pattern `"()aa()"` on the string `"flaaap"`, there will be two captures: 3 and 5.

A pattern cannot contain embedded zeros. Use `%z` instead.

5.4 Table Manipulation

This library provides generic functions for table manipulation. It provides all its functions inside the table `table`.

Most functions in the table library assume that the table represents an array or a list. For those functions, an important concept is the *size* of the array. There are three ways to specify that size:

- the field `"n"` — When the table has a field `"n"` with a numerical value, that value is assumed as its size.
- `setn` — You can call the `table.setn` function to explicitly set the size of a table.
- implicit size — Otherwise, the size of the object is one less the first integer index with a **nil** value.

For more details, see the descriptions of the `table.getn` and `table.setn` functions.

• `table.concat (table [, sep [, i [, j]]])`

Returns `table[i]..sep..table[i+1] ... sep..table[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the size of the table. If `i` is greater than `j`, returns the empty string.

• `table.foreach (table, f)`

Executes the given `f` over all elements of `table`. For each element, `f` is called with the index and respective value as arguments. If `f` returns a non-**nil** value, then the loop is broken, and this value is returned as the final value of `foreach`.

See the `next` function for extra information about table traversals.

• `table.foreachi (table, f)`

Executes the given `f` over the numerical indices of `table`. For each index, `f` is called with the index and respective value as arguments. Indices are visited in sequential order, from 1 to `n`, where `n` is the size of the table (see §5.4). If `f` returns a non-**nil** value, then the loop is broken and this value is returned as the result of `foreachi`.

• `table.getn (table)`

Returns the size of a table, when seen as a list. If the table has an `n` field with a numeric value, this value is the size of the table. Otherwise, if there was a previous call to `table.setn` over this table, the respective value is returned. Otherwise, the size is one less the first integer index with a **nil** value.

• `table.sort (table [, comp])`

Sorts table elements in a given order, *in-place*, from `table[1]` to `table[n]`, where `n` is the size of the table (see §5.4). If `comp` is given, then it must be a function that receives two table elements, and returns true when the first is less than the second (so that `not comp(a[i+1],a[i])` will be true after the sort). If `comp` is not given, then the standard Lua operator `<` is used instead.

The sort algorithm is *not* stable, that is, elements considered equal by the given order may have their relative positions changed by the sort.

• `table.insert (table, [pos,] value)`

Inserts element `value` at position `pos` in `table`, shifting up other elements to open space, if necessary. The default value for `pos` is `n+1`, where `n` is the size of the table (see §5.4), so that a call `table.insert(t,x)` inserts `x` at the end of table `t`. This function also updates the size of the table by calling `table.setn(table, n+1)`.

• `table.remove (table [, pos])`

Removes from `table` the element at position `pos`, shifting down other elements to close the space, if necessary. Returns the value of the removed element. The default value for `pos` is `n`, where `n` is the size of the table (see §5.4), so that a call `table.remove(t)` removes the last element of table `t`. This function also updates the size of the table by calling `table.setn(table, n-1)`.

• `table.setn (table, n)`

Updates the size of a table. If the table has a field `"n"` with a numerical value, that value is changed to the given `n`. Otherwise, it updates an internal state so that subsequent calls to `table.getn(table)` return `n`.

5.5 Mathematical Functions

This library is an interface to most of the functions of the standard C math library. (Some have slightly different names.) It provides all its functions inside the table `math`. In addition, it registers the global `__pow` for the binary exponentiation operator `^`, so that `x^y` returns x^y. The library provides the following functions:

```
math.abs     math.acos    math.asin    math.atan    math.atan2
math.ceil    math.cos     math.deg     math.exp     math.floor
math.log     math.log10   math.max     math.min     math.mod
math.pow     math.rad     math.sin     math.sqrt    math.tan
math.frexp   math.ldexp   math.random  math.randomseed
```

plus a variable `math.pi`. Most of them are only interfaces to the corresponding functions in the C library. All trigonometric functions work in radians (previous versions of Lua used degrees). The functions `math.deg` and `math.rad` convert between radians and degrees.

The function `math.max` returns the maximum value of its numeric arguments. Similarly, `math.min` computes the minimum. Both can be used with 1, 2, or more arguments.

The functions `math.random` and `math.randomseed` are interfaces to the simple random generator functions `rand` and `srand` that are provided by ANSI C. (No guarantees can be given for their statistical properties.) When called without arguments, `math.random` returns a pseudo-random

real number in the range $[0, 1)$. When called with a number n, `math.random` returns a pseudo-random integer in the range $[1, n]$. When called with two arguments, l and u, `math.random` returns a pseudo-random integer in the range $[l, u]$. The `math.randomseed` function sets a "seed" for the pseudo-random generator: Equal seeds produce equal sequences of numbers.

5.6 Input and Output Facilities

The I/O library provides two different styles for file manipulation. The first one uses implicit file descriptors, that is, there are operations to set a default input file and a default output file, and all input/output operations are over those default files. The second style uses explicit file descriptors.

When using implicit file descriptors, all operations are supplied by table `io`. When using explicit file descriptors, the operation `io.open` returns a file descriptor and then all operations are supplied as methods by the file descriptor.

The table `io` also provides three predefined file descriptors with their usual meanings from C: `io.stdin`, `io.stdout`, and `io.stderr`.

A file handle is a userdata containing the file stream (`FILE*`), with a distinctive metatable created by the I/O library.

Unless otherwise stated, all I/O functions return **nil** on failure (plus an error message as a second result) and some value different from **nil** on success.

• `io.close ([file])`

Equivalent to `file:close()`. Without a `file`, closes the default output file.

• `io.flush ()`

Equivalent to `file:flush` over the default output file.

• `io.input ([file])`

When called with a file name, it opens the named file (in text mode), and sets its handle as the default input file. When called with a file handle, it simply sets that file handle as the default input file. When called without parameters, it returns the current default input file.

In case of errors this function raises the error, instead of returning an error code.

• `io.lines ([filename])`

Opens the given file name in read mode and returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in io.lines(filename) do ... end
```

will iterate over all lines of the file. When the iterator function detects the end of file, it returns **nil** (to finish the loop) and automatically closes the file.

The call `io.lines()` (without a file name) is equivalent to `io.input():lines()`, that is, it iterates over the lines of the default input file.

• `io.open (filename [, mode])`

This function opens a file, in the mode specified in the string `mode`. It returns a new file handle, or, in case of errors, **nil** plus an error message.

The `mode` string can be any of the following:

"r" read mode (the default);

"w" write mode;

"a" append mode;

"r+" update mode, all previous data is preserved;

"w+" update mode, all previous data is erased;

"a+" append update mode, previous data is preserved, writing is only allowed at the end of file.

The `mode` string may also have a `b` at the end, which is needed in some systems to open the file in binary mode. This string is exactly what is used in the standard C function `fopen`.

• `io.output ([file])`

Similar to `io.input`, but operates over the default output file.

• `io.read (format1, ...)`

Equivalent to `io.input():read`.

• `io.tmpfile ()`

Returns a handle for a temporary file. This file is open in update mode and it is automatically removed when the program ends.

• `io.type (obj)`

Checks whether `obj` is a valid file handle. Returns the string `"file"` if `obj` is an open file handle, `"closed file"` if `obj` is a closed file handle, and **nil** if `obj` is not a file handle.

• `io.write (value1, ...)`

Equivalent to `io.output():write`.

• `file:close ()`

Closes `file`.

• `file:flush ()`

Saves any written data to `file`.

• `file:lines ()`

Returns an iterator function that, each time it is called, returns a new line from the file. Therefore, the construction

```
for line in file:lines() do ... end
```

will iterate over all lines of the file. (Unlike `io.lines`, this function does not close the file when the loop ends.)

• `file:read (format1, ...)`

Reads the file `file`, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or **nil** if it cannot read data with the specified format. When called without formats, it uses a default format that reads the entire next line (see below).

The available formats are

"*n" reads a number; this is the only format that returns a number instead of a string.

"*a" reads the whole file, starting at the current position. On end of file, it returns the empty string.

"*l" reads the next line (skipping the end of line), returning **nil** on end of file. This is the default format.

number reads a string with up to that number of characters, returning **nil** on end of file. If number is zero, it reads nothing and returns an empty string, or **nil** on end of file.

• `file:seek ([whence] [, offset])`

Sets and gets the file position, measured from the beginning of the file, to the position given by `offset` plus a base specified by the string `whence`, as follows:

"set" base is position 0 (beginning of the file);

"cur" base is current position;

"end" base is end of file;

In case of success, function `seek` returns the final file position, measured in bytes from the beginning of the file. If this function fails, it returns **nil**, plus a string describing the error.

The default value for `whence` is `"cur"`, and for `offset` is 0. Therefore, the call `file:seek()` returns the current file position, without changing it; the call `file:seek("set")` sets the position to the beginning of the file (and returns 0); and the call `file:seek("end")` sets the position to the end of the file, and returns its size.

• `file:write (value1, ...)`

Writes the value of each of its arguments to the filehandle `file`. The arguments must be strings or numbers. To write other values, use `tostring` or `string.format` before `write`.

5.7 Operating System Facilities

This library is implemented through table `os`.

• `os.clock ()`

Returns an approximation of the amount of CPU time used by the program, in seconds.

• `os.date ([format [, time]])`

Returns a string or a table containing date and time, formatted according to the given string `format`.

If the `time` argument is present, this is the time to be formatted (see the `os.time` function for a description of this value). Otherwise, `date` formats the current time.

If `format` starts with '`!`', then the date is formatted in Coordinated Universal Time. After that optional character, if `format` is `*t`, then `date` returns a table with the following fields: `year` (four digits), `month` (1–12), `day` (1–31), `hour` (0–23), `min` (0–59), `sec` (0–61), `wday` (weekday, Sunday is 1), `yday` (day of the year), and `isdst` (daylight saving flag, a boolean).

If `format` is not `*t`, then `date` returns the date as a string, formatted according with the same rules as the C function `strftime`.

When called without arguments, `date` returns a reasonable date and time representation that depends on the host system and on the current locale (that is, `os.date()` is equivalent to `os.date("%c")`).

• `os.difftime (t2, t1)`

Returns the number of seconds from time `t1` to time `t2`. In Posix, Windows, and some other systems, this value is exactly `t2`−`t1`.

• `os.execute (command)`

This function is equivalent to the C function `system`. It passes `command` to be executed by an operating system shell. It returns a status code, which is system-dependent.

• `os.exit ([code])`

Calls the C function `exit`, with an optional `code`, to terminate the host program. The default value for `code` is the success code.

• `os.getenv (varname)`

Returns the value of the process environment variable `varname`, or **nil** if the variable is not defined.

• `os.remove (filename)`

Deletes the file with the given name. If this function fails, it returns **nil**, plus a string describing the error.

• `os.rename (oldname, newname)`

Renames file named `oldname` to `newname`. If this function fails, it returns **nil**, plus a string describing the error.

• `os.setlocale (locale [, category])`

Sets the current locale of the program. `locale` is a string specifying a locale; `category` is an optional string describing which category to change: `"all"`, `"collate"`, `"ctype"`, `"monetary"`, `"numeric"`, or `"time"`; the default category is `"all"`. The function returns the name of the new locale, or **nil** if the request cannot be honored.

• `os.time ([table])`

Returns the current time when called without arguments, or a time representing the date and time specified by the given table. This table must have fields `year`, `month`, and `day`, and may have fields `hour`, `min`, `sec`, and `isdst` (for a description of these fields, see the `os.date` function).

The returned value is a number, whose meaning depends on your system. In Posix, Windows, and some other systems, this number counts the number of seconds since some given start time (the "epoch"). In other systems, the meaning is not specified, and the number returned by `time` can be used only as an argument to `date` and `difftime`.

• `os.tmpname ()`

Returns a string with a file name that can be used for a temporary file. The file must be explicitly opened before its use and removed when no longer needed.

This function is equivalent to the `tmpnam` C function, and many people (and even some compilers!) advise against its use, because between the time you call this function and the time you open the file, it is possible for another process to create a file with the same name.

5.8 The Reflexive Debug Interface

The `debug` library provides the functionality of the debug interface to Lua programs. You should exert care when using this library. The functions provided here should be used exclusively for debugging and similar tasks, such as profiling. Please resist the temptation to use them as a usual programming tool: They can be very slow. Moreover, `setlocal` and `getlocal` violate the privacy of local variables and therefore can compromise some otherwise secure code.

All functions in this library are provided inside a `debug` table.

• `debug.debug ()`

Enters an interactive mode with the user, running each string that the user enters. Using simple commands and other debug facilities, the user can inspect global and local variables, change their values, evaluate expressions, and so on. A line containing only the word `cont` finishes this function, so that the caller continues its execution.

Note that commands for `debug.debug` are not lexically nested with any function, so they have no direct access to local variables.

• `debug.gethook ()`

Returns the current hook settings, as three values: the current hook function, the current hook mask, and the current hook count (as set by the `debug.sethook` function).

• `debug.getinfo (function [, what])`

This function returns a table with information about a function. You can give the function directly, or you can give a number as the value of `function`, which means the function running at level `function` of the call stack: Level 0 is the current function (`getinfo` itself); level 1 is the function that called `getinfo`; and so on. If `function` is a number larger than the number of active functions, then `getinfo` returns **nil**.

The returned table contains all the fields returned by `lua_getinfo`, with the string `what` describing which fields to fill in. The default for `what` is to get all information available. If present, the option '`f`' adds a field named `func` with the function itself.

For instance, the expression `debug.getinfo(1,"n").name` returns the name of the current function, if a reasonable name can be found, and `debug.getinfo(print)` returns a table with all available information about the `print` function.

• `debug.getlocal (level, local)`

This function returns the name and the value of the local variable with index `local` of the function at level `level` of the stack. (The first parameter or local variable has index 1, and so on, until the last active local variable.) The function returns **nil** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `debug.getinfo` to check whether the level is valid.)

• `debug.getupvalue (func, up)`

This function returns the name and the value of the upvalue with index `up` of the function `func`. The function returns **nil** if there is no upvalue with the given index.

• `debug.setlocal (level, local, value)`

This function assigns the value `value` to the local variable with index `local` of the function at level `level` of the stack. The function returns **nil** if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `getinfo` to check whether the level is valid.)

• `debug.setupvalue (func, up, value)`

This function assigns the value `value` to the upvalue with index `up` of the function `func`. The function returns **nil** if there is no upvalue with the given index.

• `debug.sethook (hook, mask [, count])`

Sets the given function as a hook. The string `mask` and the number `count` describe when the hook will be called. The string mask may have the following characters, with the given meaning:

- `"c"` The hook is called every time Lua calls a function;
- `"r"` The hook is called every time Lua returns from a function;
- `"l"` The hook is called every time Lua enters a new line of code.

With a `count` different from zero, the hook is called after every `count` instructions.

When called without arguments, the `debug.sethook` function turns off the hook.

When the hook is called, its first parameter is always a string describing the event that triggered its call: `"call"`, `"return"` (or `"tail return"`), `"line"`, and `"count"`. Moreover, for line events, it also gets as its second parameter the new line number. Inside a hook, you can call `getinfo` with level 2 to get more information about the running function (level 0 is the `getinfo` function, and level 1 is the hook function), unless the event is `"tail return"`. In this case, Lua is only simulating the return, and a call to `getinfo` will return invalid data.

• `debug.traceback ([message])`

Returns a string with a traceback of the call stack. An optional `message` string is appended at the beginning of the traceback. This function is typically used with `xpcall` to produce better error messages.

6 Lua Stand-alone

Although Lua has been designed as an extension language, to be embedded in a host C program, it is also frequently used as a stand-alone language. An interpreter for Lua as a stand-alone language, called simply `lua`, is provided with the standard distribution. The stand-alone interpreter includes all standard libraries plus the reflexive debug interface. Its usage is:

```
lua [options] [script [args]]
```

The options are:

- `-` executes `stdin` as a file;
- `-e` ***stat*** executes string *stat*;
- `-l` ***file*** "requires" *file*;
- `-i` enters interactive mode after running *script*;
- `-v` prints version information;
- `--` stop handling options.

After handling its options, `lua` runs the given *script*, passing to it the given *args*. When called without arguments, `lua` behaves as `lua -v -i` when `stdin` is a terminal, and as `lua -` otherwise.

Before running any argument, the interpreter checks for an environment variable `LUA_INIT`. If its format is `@`*filename*, then lua executes the file. Otherwise, lua executes the string itself.

All options are handled in order, except `-i`. For instance, an invocation like

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

will first set `a` to 1, then print `a`, and finally run the file `script.lua`. (Here, `$` is the shell prompt. Your prompt may be different.)

Before starting to run the script, `lua` collects all arguments in the command line in a global table called `arg`. The script name is stored in index 0, the first argument after the script name goes to index 1, and so on. The field `n` gets the number of arguments after the script name. Any arguments before the script name (that is, the interpreter name plus the options) go to negative indices. For instance, in the call

```
$ lua -la.lua b.lua t1 t2
```

the interpreter first runs the file `a.lua`, then creates a table

```
arg = { [-2] = "lua", [-1] = "-la.lua", [0] = "b.lua",
        [1] = "t1", [2] = "t2"; n = 2 }
```

and finally runs the file `b.lua`.

In interactive mode, if you write an incomplete statement, the interpreter waits for its completion.

If the global variable `_PROMPT` is defined as a string, then its value is used as the prompt. Therefore, the prompt can be changed directly on the command line:

```
$ lua -e"_PROMPT='myprompt> '" -i
```

(the outer pair of quotes is for the shell, the inner is for Lua), or in any Lua programs by assigning to `_PROMPT`. Note the use of `-i` to enter interactive mode; otherwise, the program would end just after the assignment to `_PROMPT`.

In Unix systems, Lua scripts can be made into executable programs by using `chmod +x` and the `#!` form, as in

```
#!/usr/local/bin/lua
```

(Of course, the location of the Lua interpreter may be different in your machine. If `lua` is in your `PATH`, then

```
#!/usr/bin/env lua
```

is a more portable solution.)

Acknowledgments

The Lua team is grateful to Tecgraf for its continued support to Lua. We thank everyone at Tecgraf, specially the head of the group, Marcelo Gattass. At the risk of omitting several names, we also thank the following individuals for supporting, contributing to, and spreading the word about Lua: Alan Watson. André Clinio, André Costa, Antonio Scuri, Asko Kauppi, Bret Mogilefsky, Cameron Laird, Carlos Cassino, Carlos Henrique Levy, Claudio Terra, David Jeske, Ed Ferguson, Edgar Toernig, Erik Hougaard, Jim Mathies, John Belmonte, John Passaniti, John Roll, Jon Erickson, Jon Kleiser, Mark Ian Barlow, Nick Trout, Noemi Rodriguez, Norman Ramsey, Philippe Lhoste, Renata Ratton, Renato Borges, Renato Cerqueira, Reuben Thomas, Stephan Herrmann, Steve Dekorte, Thatcher Ulrich, Tomás Gorham, Vincent Penquerc'h. Thank you!

Incompatibilities with Previous Versions

Lua 5.0 is a major release. There are several incompatibilities with its previous version, Lua 4.0.

Incompatibilities with version 4.0

Changes in the Language

- The whole tag-method scheme was replaced by metatables.
- Function calls written between parentheses result in exactly one value.
- A function call as the last expression in a list constructor (like `{a,b,f()}`) has all its return values inserted in the list.
- The precedence of **or** is smaller than the precedence of **and**.
- **in**, **false**, and **true** are reserved words.
- The old construction `for k,v in t`, where `t` is a table, is deprecated (although it is still supported). Use `for k,v in pairs(t)` instead.
- When a literal string of the form `[[...]]` starts with a newline, this newline is ignored.
- Upvalues in the form `%var` are obsolete; use external local variables instead.

Changes in the Libraries

- Most library functions now are defined inside tables. There is a compatibility script (`compat.lua`) that redefine most of them as global names.
- In the math library, angles are expressed in radians. With the compatibility script (`compat.lua`), functions still work in degrees.
- The `call` function is deprecated. Use `f(unpack(tab))` instead of `call(f, tab)` for unprotected calls, or the new `pcall` function for protected calls.
- `dofile` do not handle errors, but simply propagates them.
- `dostring` is deprecated. Use `loadstring` instead.
- The `read` option `*w` is obsolete.
- The `format` option `%n$` is obsolete.

Changes in the API

- `lua_open` does not have a stack size as its argument (stacks are dynamic).
- `lua_pushuserdata` is deprecated. Use `lua_newuserdata` or `lua_pushlightuserdata` instead.

The Complete Syntax of Lua

chunk → { *stat* [';'] }
block → *chunk*
stat → *varlist1* '=' *explist1*
| *functioncall*
| **do** *block* **end**
| **while** *exp* **do** *block* **end**
| **repeat** *block* **until** *exp*
| **if** *exp* **then** *block* { **elseif** *exp* **then** *block* } [**else** *block*] **end**
| **return** [*explist1*]
| **break**
| **for** `Name` '=' *exp* ',' *exp* [',' *exp*] **do** *block* **end**
| **for** `Name` { ',' `Name` } **in** *explist1* **do** *block* **end**
| **function** *funcname funcbody*
| **local function** `Name` *funcbody*
| **local** *namelist* [*init*]
funcname → `Name` { '.' `Name` } [':' `Name`]
varlist1 → *var* { ',' *var* }
var → `Name` | *prefixexp* '[' *exp* ']' | *prefixexp* '.' `Name`
namelist → `Name` { ',' `Name` }
init → '=' *explist1*
explist1 → { *exp* ',' } *exp*
exp → **nil** | **false** | **true** | `Number` | `Literal`
| *function* | *prefixexp* | *tableconstructor* | *exp binop exp* | *unop exp*
prefixexp → *var* | *functioncall* | '(' *exp* ')'
functioncall → *prefixexp args* | *prefixexp* ':' `Name` *args*
args → '(' [*explist1*] ')' | *tableconstructor* | `Literal`
function → **function** *funcbody*
funcbody → '(' [*parlist1*] ')' *block* **end**
parlist1 → `Name` { ',' `Name` } [',' '...'] | '...'
tableconstructor → '{' [*fieldlist*] '}'
fieldlist → *field* { *fieldsep field* } [*fieldsep*]
field → '[' *exp* ']' '=' *exp* | *name* '=' *exp* | *exp*
fieldsep → ',' | ';'
binop → '+' | '-' | '*' | '/' | '^' | '..'
| '<' | '<=' | '>' | '>=' | '==' | '~='
| **and** | **or**
unop → '-' | **not**

Index

_G 39
_LOADED 42
_PROMPT 58
_REQUIREDNAME 42
_VERSION 43

A

acceptable index 23
"add" event 15
adjustment 5
and 9
arg 13
arguments 13
arithmetic operators 8
arrays 3
assert 39
assignment 5
associative arrays 3

B

basic types 3
block 5
boolean 3
break statement 6

C

C API 22
  Chunkreader 28
  lua_call 30
  lua_CFunction 32
  lua_checkstack 23
  lua_close 22
  lua_concat 27
  lua_cpcall 34
  lua_Debug 35
  lua_equal 25
  LUA_ERRERR 31
  LUA_ERRMEM 28, 31
  lua_error 34
  LUA_ERRRUN 31
  LUA_ERRSYNTAX 28
  lua_getfenv 30
  lua_getgccount 27
  lua_getgcthreshold 27
lua_gethook 38
lua_gethookcount 38
lua_gethookmask 38
lua_getinfo 35
lua_getlocal 36
lua_getmetatable 28
lua_getstack 35
lua_gettable 29
lua_gettop 23
lua_getupvalue 36
LUA_GLOBALSINDEX 30
lua_Hook 37
LUA_HOOKCALL 37
LUA_HOOKCOUNT 37
LUA_HOOKLINE 37
LUA_HOOKRET 37
LUA_HOOKTAILRET 37
lua_insert 24
lua_isboolean 24
lua_iscfunction 24
lua_isfunction 24
lua_islightuserdata 24
lua_isnil 24
lua_isnumber 24
lua_isstring 24
lua_istable 24
lua_isuserdata 24
lua_lessthan 25
lua_load 28
LUA_MASKCALL 37
LUA_MASKCOUNT 37
LUA_MASKLINE 37
LUA_MASKRET 37
LUA_MINSTACK 23
LUA_MULTRET 30
lua_newtable 29
lua_newthread 34
lua_newuserdata 27
lua_next 29
lua_Number 25
lua_open 22
lua_pop 24
lua_pushboolean 26
lua_pushcclosure 33

lua_pushcfunction 26
lua_pushfstring 26
lua_pushlightuserdata 26
lua_pushlstring 26
lua_pushnil 26
lua_pushnumber 26
lua_pushstring 26
lua_pushvalue 24
lua_pushvfstring 26
lua_rawequal 25
lua_rawget 29
lua_rawgeti 30
lua_rawset 29
lua_rawseti 30
lua_register 32
LUA_REGISTRYINDEX 33
lua_remove 24
lua_replace 24
lua_resume 34
lua_setfenv 30
lua_setgcthreshold 27
lua_sethook 37
lua_setlocal 36
lua_setmetatable 28
lua_settable 29
lua_settop 24
lua_setupvalue 36
lua_State 22
lua_strlen 25
lua_toboolean 25
lua_tocfunction 25
lua_tonumber 25
lua_topointer 25
lua_tostring 25
lua_tothread 25
lua_touserdata 25
lua_type 24
lua_typename 25
lua_upvalueindex 33
lua_xmove 34
lua_yield 34
luaopen_base 38
luaopen_debug 38
luaopen_io 38
luaopen_math 38
luaopen_string 38
luaopen_table 38
C closure 33
"call" event 19
captures 48
character class 47
chunk 5
collectgarbage 39
comments 3
concatenation 10
"concatenation" event 17
condition expression 6
constructors 10
coroutine 43
coroutine.create 21, 43
coroutine.resume 21, 44
coroutine.status 44
coroutine.wrap 21, 44
coroutine.yield 21, 44

D

debug 55
debug.debug 55
debug.gethook 55
debug.getinfo 56
debug.getlocal 56
debug.getupvalue 56
debug.sethook 56
debug.setlocal 56
debug.setupvalue 56
debug.traceback 57
"div" event 16
dofile 39

E

eight-bit clean 3
environment tables 4
"eq" event 17
error 39
events 15
exponentiation 9
expressions 8

F

file:close 52
file:flush 52
file:lines 53
file:read 53
file:seek 53
file:write 53

finalizers 20
for statement 6
full userdata 27
function 3
function call 11
function definitions 12

G

garbage collector 20
`gcinfo` 40
`getfenv` 39
`getmetatable` 40
grammar 60
- args 11, 60
- binop 60
- block 5, 60
- chunk 5, 60
- exp 8, 60
- explist1 ..... 5, 60
- field 10, 60
- fieldlist 10, 60
- fieldsep 10, 60
- funcbody 12, 60
- funcname 12, 60
- function 12, 60
- functioncall 11, 60
- init 60
- namelist 8, 60
- parlist1 ..... 13, 60
- prefixexp 8, 60
- stat 5–8, 12, 60
- tableconstructor 10, 60
- unop 60
- var 4, 60
- varlist1 ..... 5, 60

I

identifiers 1
if-then-else statement 6
"index" event 18, 19
`io` 51
`io.close` 51
`io.flush` 51
`io.input` 51
`io.lines` 51
`io.open` 51, 52
`io.output` 52
`io.read` 52
`io.stderr` 51
`io.stdin` 51
`io.stdout` 51
`io.tmpfile` 52
`io.type` 52
`io.write` 52
`ipairs` 40
iterators 7

K

keywords
- `and` 9
- `break` 6
- `do` 5
- `else` 6
- `elseif` 6
- `false` 8
- `for` 6
- `function` 12
- `if` 6
- `in` 6
- `local` 8
- `nil` 8
- `not` 9
- `or` 9
- `repeat` 6
- `return` 6
- `true` 8
- `until` 6
- `while` 6

keywords 1

L

lexical scoping 14
light userdata 27
literal strings 2
`loadfile` 40
`loadlib` 40
`loadstring` 40
local variables 4, 8
logical operators 9
"lt" event 18
lua stand-alone 57
`LUA_INIT` 57
`LUA_PATH` 42
`luac` 5, 28

M

`math` 50
`math.abs` 50
`math.acos` 50
`math.asin` 50
`math.atan` 50
`math.atan2` 50
`math.ceil` 50
`math.cos` 50
`math.def` 50
`math.exp` 50
`math.floor` 50
`math.frexp` 50
`math.ldexp` 50
`math.log` 50
`math.log10` 50
`math.max` 50
`math.min` 50
`math.mod` 50
`math.pi` 50
`math.pow` 50
`math.rad` 50
`math.random` 50
`math.randomseed` 50
`math.sin` 50
`math.sqrt` 50
`math.tan` 50
metamethod
  add 15
  call 19
  concatenation 17
  div 16
  eq 17
  index 18, 19
  lt 18
  mul 16
  pow 16
  sub 16
  unm 16
metamethod 15
metamethods 15
metatable 15
methods 13
"mul" event 16
multiple assignment 5

N

`next` 41
nil 3
not 9
number 3
numerical constants 3

O

operator precedence 10
or 9
`os` 53
`os.clock` 54
`os.date` 54
`os.difftime` 54
`os.execute` 54
`os.exit` 54
`os.getenv` 54
`os.remove` 54
`os.rename` 54
`os.setlocale` 55
`os.time` 55
`os.tmpname` 55

P

`pairs` 41
panic function 33
pattern 48
pattern item 48
`pcall` 41
"pow" event 16
pre-compilation 5
`print` 41
proper tail calls 12
pseudo-indices 23

R

`rawequal` 41
`rawget` 41
`rawset` 41
records 3
relational operators 9
repeat-until statement 6
`require` 42
reserved words 1
return statement 6

S

`self` 13

`setfenv` 42
`setmetatable` 42
short-cut evaluation 9
stack index 23
state 22
statements 4
string 3
`string` 44
`string.byte` 44
`string.char` 44
`string.dump` 45
`string.find` 45
`string.format` 45
`string.gsub` 46
`string.len` 45
`string.lower` 45
`string.rep` 45
`string.sub` 45
`string.upper` 45
"sub" event 16

T

table 3
`table` 49
`table.concat` 49
`table.foreach` 49
`table.foreachi` 49
`table.getn` 49
`table.insert` 50
`table.remove` 50
`table.setn` 50
`table.sort` 50
tail call 12
thread 3
tokens 2
`tonumber` 42
`tostring` 43
`type` 43

U

"unm" event 16
`unpack` 43
upvalue 14
userdata 3

V

valid index 23
values and types 3
vararg function 13
version 4.0 59
visibility 14

W

weak references 20
weak table 20
while-do statement 6

X

`xpcall` 43