The Programming Language Lua

Reference Manual for Lua version 4.0
Last revised on November 6, 2000

The Lua language and this implementation have been entirely designed and written by Waldemar Celes, Roberto Ierusalimschy, and Luiz Henrique de Figueiredo at TeCGraf, PUC-Rio in Brazil.

This implementation contains no third-party code.

Copies of this manual can be obtained at `http://www.tecgraf.puc-rio.br/lua/`.

Reference Manual of the Programming Language Lua 4.0

Roberto Ierusalimschy   Luiz Henrique de Figueiredo   Waldemar Celes

lua@tecgraf.puc-rio.br

TeCGraf — Computer Science Department — PUC-Rio

$Date: 2000/10/31 18:20:01

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 bytecodes, and has automatic memory management with garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

This document describes version 4.0 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 *bytecodes*, 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 4.0 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

Lua is an extension programming language designed to support general procedural programming with data description facilities. 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 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. This host program can invoke functions to execute a piece of code in Lua, 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.

Lua is free-distribution software, and is provided as usual with no guarantees, as stated in its copyright notice. The implementation described in this manual is available at the following URL's:

```
http://www.tecgraf.puc-rio.br/lua/
ftp://ftp.tecgraf.puc-rio.br/pub/lua/
```

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 the web site above.

- 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.

2 Environment and Chunks

All statements in Lua are executed in a *global environment*. This environment is initialized with a call from the embedding program to `lua_open` and persists until a call to `lua_close`, or the end of the embedding program. If necessary, the host programmer can create multiple independent global environments, and freely switch between them (see §5.1).

The global environment can be manipulated by Lua code or by the embedding program, which can read and write global variables using API functions from the library that implements Lua.

Global variables in Lua do not need to be declared. Any variable is assumed to be global unless explicitly declared local (see §4.4.6). Before the first assignment, the value of a global variable is **nil** (this default can be changed; see §4.8). A table is used to keep all global names and values (tables are explained in §3).

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:

$$chunk \quad \rightarrow \quad \{stat\ [\text{`}\texttt{;}\text{'}]\}$$

Statements are described in §4.4. (The notation above is the usual extended BNF, in which $\{a\}$ means 0 or more a's, $[a]$ means an optional a, and $\{a\}^+$ means one or more a's. The complete syntax of Lua is given on page 49.)

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 bytecodes for a virtual machine, and then the statements are executed in sequential order, by simulating the virtual machine. All modifications a chunk effects on the global environment persist after the chunk ends.

Chunks may also be pre-compiled into binary form and stored in files; see program `luac` for details. Text files with chunks and their binary pre-compiled forms are interchangeable. Lua automatically detects the file type and acts accordingly.

3 Types and Tags

Lua is a *dynamically typed language.* This means that variables do not have types; only values do. Therefore, there are no type definitions in the language. All values carry their own type. Besides a type, all values also have a *tag*.

There are six basic types in Lua: *nil*, *number*, *string*, *function*, *userdata*, and *table*. *Nil* is the type of the value **nil**, whose main property is to be different from any other value. *Number* represents real (double-precision floating-point) numbers, while *string* has the usual meaning. Lua is 8-bit clean, and so strings may contain any 8-bit character, including embedded zeros (`'\0'`) (see §4.1). The `type` function returns a string describing the type of a given value (see §6.1).

Functions are considered *first-class values* in Lua. This 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. The two kinds of functions can be distinguished by their tags: all Lua functions have the same tag, and all C functions have the same tag, which is different from the tag of Lua functions. The `tag` function returns the tag of a given value (see §6.1).

The type *userdata* is provided to allow arbitrary C pointers to be stored in Lua variables. This type corresponds to a `void*` and has no pre-defined operations in Lua, except assignment and equality test. However, by using *tag methods*, the programmer can define operations for *userdata* values (see §4.8).

The type *table* implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any value (except **nil**). Therefore, this type may be used not only to represent ordinary arrays, but also symbol tables, sets, records, graphs, trees, etc. Tables are the main data structuring mechanism in Lua. 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"]`. Tables may also carry *methods*: Because functions are first class values, table fields may contain functions. The form `t:f(x)` is syntactic sugar for `t.f(t,x)`, which calls the method `f` from the table `t` passing the table itself as the first parameter (see §4.5.9).

Note that tables are *objects*, and not values. Variables do not contain tables, only *references* to them. Assignment, parameter passing, and returns always manipulate references to tables, and do not imply any kind of copy. Moreover, tables must be explicitly created before used (see §4.5.7).

Each of the types *nil*, *number*, and *string* has a different tag. All values of each of these types have the same pre-defined tag. As explained above, values of type *function* can have two different tags, depending on whether they are Lua functions or C functions. Finally, values of type *userdata* and *table* can have variable tags, assigned by the programmer (see §4.8). The `tag` function returns the tag of a given value. User tags are created with the function `newtag`. The `settag` function is used to change the tag of a table (see §6.1). The tag of userdata values can only be set from C (see §5.7). Tags are mainly used to select *tag methods* when some events occur. Tag methods are the main mechanism for extending the semantics of Lua (see §4.8).

4 The Language

This section describes the lexis, the syntax, and the semantics of Lua.

4.1 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, except that 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 words are *reserved*, and cannot be used as identifiers:

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

Lua is a case-sensitive language: `and` is a reserved word, but `And` and `ánd` (if the locale permits) are two different, valid identifiers. As a convention, identifiers starting with underscore followed by uppercase letters (such as `_INPUT`) are reserved for internal variables.

The following strings denote other tokens:

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

Literal strings can be delimited by matching single or double quotes, and can contain the C-like escape sequences '`\a`' (bell), '`\b`' (backspace), '`\f`' (form feed), '`\n`' (newline), '`\r`' (carriage return), '`\t`' (horizontal tab), '`\v`' (vertical tab), '`\\`' (backslash), '`\"`' (double quote), '`\'`' (single quote), and '`\`*newline*' (that is, a backslash followed by a real newline, which results in a newline in the string). A character in a string may also be specified by its numerical value, through 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 '`\000`'.

Literal strings can also be delimited by matching `[[` ... `]]`. Literals in this bracketed form may run for several lines, may contain nested `[[` ... `]]` pairs, and do not interpret escape sequences. This form is specially convenient for writing strings that contain program pieces or other quoted strings. As an example, in a system using ASCII, the following three literals are equivalent:

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

Comments start anywhere outside a string with a double hyphen (`--`) and run until the end of the line. Moreover, 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 §8).

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
```

4.2 Coercion

Lua provides some automatic conversions between 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 when a string is expected, that number is converted to a string, in a reasonable format. The format is chosen so that a conversion from number to string then back to number reproduces the original number *exactly*. Thus, the conversion does not necessarily produces nice-looking text for some numbers. For complete control of how numbers are converted to strings, use the `format` function (see §6.2).

4.3 Adjustment

Functions in Lua can return many values. Because there are no type declarations, when a function is called the system does not know how many values the function will return, or how many parameters it needs. Therefore, sometimes, a list of values must be *adjusted*, at run time, to a given length. If there are more values than are needed, then the excess values are thrown away. If there are less values than are needed, then the list is extended with as many **nil**'s as needed. This adjustment occurs in multiple assignments (see §4.4.2) and in function calls (see §4.5.8).

4.4 Statements

Lua supports an almost conventional set of statements, similar to those in Pascal or C. The conventional commands include assignment, control structures, and procedure calls. Non-conventional commands include table constructors (see §4.5.7) and local variable declarations (see §4.4.6).

4.4.1 Blocks

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

$$\textit{block} \quad \rightarrow \quad \textit{chunk}$$

A block may be explicitly delimited:

$$\textit{stat} \quad \rightarrow \quad \textbf{do}\ \textit{block}\ \textbf{end}$$

Explicit blocks are useful to control the scope of local variables (see §4.4.6). Explicit blocks are also sometimes used to add a **return** or **break** statement in the middle of another block (see §4.4.3).

4.4.2 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:

$$\begin{aligned} \textit{stat} &\quad \rightarrow \quad \textit{varlist1}\ \text{`}\texttt{=}\text{'}\ \textit{explist1} \\ \textit{varlist1} &\quad \rightarrow \quad \textit{var}\ \{\text{`}\texttt{,}\text{'}\ \textit{var}\} \end{aligned}$$

This statement first evaluates all values on the right side and eventual indices on the left side, and then makes the assignments. So, the code

```
i = 3
i, a[i] = 4, 20
```

sets `a[3]` to 20, but does not affect `a[4]` because the `i` in `a[i]` is evaluated before it is assigned `4`.

Multiple assignment can be used to exchange two values, as in

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

The two lists in a multiple assignment may have different lengths. Before the assignment, the list of values is adjusted to the length of the list of variables (see §4.3).

A single name can denote a global variable, a local variable, or a formal parameter:

var → *name*

Square brackets are used to index a table:

var → *varorfunc* '`[`' *exp1* '`]`'
varorfunc → *var* | *functioncall*

The *varorfunc* should result in a table value, from where the field indexed by the expression *exp1* value gets the assigned value.

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

var → *varorfunc* '`.`' *name*

The meaning of assignments and evaluations of global variables and indexed variables can be changed by tag methods (see §4.8). Actually, an assignment `x = val`, where `x` is a global variable, is equivalent to a call `setglobal("x", val)` and an assignment `t[i] = val` is equivalent to `settable_event(t,i,val)`. See §4.8 for a complete description of these functions (`setglobal` is in the basic library; `settable_event` is used for explanatory purposes only).

4.4.3 Control Structures

The control structures **if**, **while**, and **repeat** have the usual meaning and familiar syntax

stat → **while** *exp1* **do** *block* **end**
stat → **repeat** *block* **until** *exp1*
stat → **if** *exp1* **then** *block* {**elseif** *exp1* **then** *block*} [**else** *block*] **end**

The condition expression *exp1* of a control structure may return any value. All values different from **nil** are considered true; only **nil** is considered false.

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

stat → **return** [*explist1*]

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

stat → **break**

A **break** ends the innermost enclosing loop (**while**, **repeat**, or **for**).

NOTE: For syntactic reasons, **return** and **break** statements can only be written as the *last* statements of a block. If it is really necessary to **return** or **break** in the middle of a block, an explicit inner block can used, as in the idiom '`do return end`', because now **return** is last statement in the inner block.

4.4.4 For Statement

The **for** statement has two forms, one for numbers and one for tables.

The numerical **for** loop has the following syntax:

stat → **for** *name* '=' *exp1* ',' *exp1* [',' *exp1*] **do** *block* **end**

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:

- `_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.
- Both the limit and the step are evaluated only once, before the loop starts.
- The variable `var` is local to the statement; you cannot use its value after the **for** ends.
- You can use **break** to exit a **for**. If you need the value of the index, assign it to another variable before breaking.

The table **for** statement traverses all pairs (index,value) of a given table. It has the following syntax:

stat → **for** *name* ',' *name* **in** *exp1* **do** *block* **end**

A **for** statement like

```
for index, value in exp do block end
```

is equivalent to the code:

```
do
  local _t = exp
  local index, value = next(t, nil)
  while index do
    block
    index, value = next(t, index)
  end
end
```

Note the following:

- `_t` is an invisible variable. The name is here for explanatory purposes only.

- The behavior is *undefined* if you assign to `index` inside the block.
- The behavior is *undefined* if you change the table `_t` during the traversal.
- The variables `index` and `value` are local to the statement; you cannot use their values after the **for** ends.
- You can use **break** to exit a **for**. If you need the value of `index` or `value`, assign them to other variables before breaking.
- The order that table elements are traversed is undefined, *even for numerical indices.* If you want to traverse indices in numerical order, use a numerical **for**.

4.4.5 Function Calls as Statements

Because of 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 §4.5.8.

4.4.6 Local Declarations

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

stat → **local** *declist* [*init*]
declist → *name* {'`,`' *name*}
init → '`=`' *explist1*

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

A chunk is also a block, and so local variables can be declared outside any explicit block.

The scope of local variables begins *after* the declaration and lasts until the end of the block. Thus, the code `local print=print` creates a local variable called `print` whose initial value is that of the *global* variable of the same name.

4.5 Expressions

4.5.1 Basic Expressions

The basic expressions in Lua are

exp → '`(`' *exp* '`)`'
exp → **nil**
exp → *number*
exp → *literal*
exp → *var*
exp → *upvalue*
exp → *function*
exp → *functioncall*
exp → *tableconstructor*

Numbers (numerical constants) and literal strings are explained in §4.1; variables are explained in §4.4.2; upvalues are explained in §4.6; function definitions are explained in §4.5.9; function calls are explained in §4.5.8. Table constructors are explained in §4.5.7.

An access to a global variable `x` is equivalent to a call `getglobal("x")` and an access to an indexed variable `t[i]` is equivalent to a call `gettable_event(t,i)`. See §4.8 for a description of these functions (`getglobal` is in the basic library; `gettable_event` is used for explanatory purposes only).

The non-terminal *exp1* is used to indicate that the values returned by an expression must be adjusted to one single value:

$$\textit{exp1} \quad \rightarrow \quad \textit{exp}$$

4.5.2 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 (according to the rules given in §4.2), then all operations except exponentiation have the usual meaning. Otherwise, an appropriate tag method is called (see §4.8). An exponentiation always calls a tag method. The standard mathematical library redefines this method for numbers, giving the expected meaning to exponentiation (see §6.3).

4.5.3 Relational Operators

The relational operators in Lua are

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

These operators return **nil** as false and a value different from **nil** as true.

Equality (`==`) first compares the tags of its operands. If they are different, then the result is **nil**. Otherwise, their values are compared. Numbers and strings are compared in the usual way. Tables, userdata, and functions are compared by reference, that is, two tables are considered equal only if they are the *same* table. The operator `~=` is exactly the negation of equality (`==`).

NOTE: The conversion rules of §4.2 *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 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 using lexicographical order. Otherwise, the "lt" tag method is called (see §4.8).

4.5.4 Logical Operators

The logical operators in Lua are

```
and   or   not
```

Like the control structures, all logical operators consider **nil** as false and anything else as true.

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

There are two useful Lua idioms that use logical operators. The first idiom is

```
x = x or v
```

which is equivalent to

```
       if x == nil then x = v end
```

This idiom sets `x` to a default value `v` when `x` is not set.

The second idiom is

```
       x = a and b or c
```

which should be read as `x = (a and b) or c`. This idiom is equivalent to

```
       if a then x = b else x = c end
```

provided that `b` is not **nil**.

4.5.5 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 in §4.2. Otherwise, the "concat" tag method is called (see §4.8).

4.5.6 Precedence

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

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

All binary operators are left associative, except for ^ (exponentiation), which is right associative.

NOTE: The pre-compiler may rearrange the order of evaluation of associative operators (such as `..` or `+`), as long as these optimizations do not change normal results. However, these optimizations may change some results if you define non-associative tag methods for these operators.

4.5.7 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 → *lfieldlist* | *ffieldlist* | *lfieldlist* ';' *ffieldlist* | *ffieldlist* ';' *lfieldlist*
lfieldlist → [*lfieldlist1*]
ffieldlist → [*ffieldlist1*]

The form *lfieldlist1* is used to initialize lists:

lfieldlist1 → *exp* {',' *exp*} [',']

The expressions in the list are assigned to consecutive numerical indices, starting with 1. For example,

```
a = {"v1", "v2", 34}
```

is equivalent to

```
do
  local temp = {}
  temp[1] = "v1"
  temp[2] = "v2"
  temp[3] = 34
  a = temp
end
```

The form *ffieldlist1* initializes other fields in a table:

ffieldlist1 → *ffield* {'`,`' *ffield*} ['`,`']
ffield → '`[`' *exp* '`]`' '`=`' *exp* | *name* '`=`' *exp*

For example,

```
a = {[f(k)] = g(y), x = 1, y = 3, [0] = b+c}
```

is equivalent to

```
do
  local temp = {}
  temp[f(k)] = g(y)
  temp.x = 1    -- or temp["x"] = 1
  temp.y = 3    -- or temp["y"] = 3
  temp[0] = b+c
  a = temp
end
```

An expression like `{x = 1, y = 4}` is in fact syntactic sugar for `{["x"] = 1, ["y"] = 4}`.

Both forms may have an optional trailing comma, and can be used in the same constructor separated by a semi-colon. For example, all forms below are correct.

```
x = {;}
x = {"a", "b",}
x = {type="list"; "a", "b"}
x = {f(0), f(1), f(2),; n=3,}
```

4.5.8 Function Calls

A function call in Lua has the following syntax:

functioncall → *varorfunc args*

First, *varorfunc* is evaluated. If its value has type *function*, then this function is called, with the given arguments. Otherwise, the "function" tag method is called, having as first parameter the value of *varorfunc*, and then the original call arguments (see §4.8).

The form

functioncall → *varorfunc* '`:`' *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:

$$\begin{array}{rcl} \mathit{args} & \to & \text{`('}\ [\mathit{explist1}]\ \text{`)'} \\ \mathit{args} & \to & \mathit{tableconstructor} \\ \mathit{args} & \to & \mathit{literal} \\ \mathit{explist1} & \to & \{\mathit{exp1}\ \text{`,'}\}\ \mathit{exp} \end{array}$$

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 §4.4.3), the number of results must be adjusted before they are used (see §4.3). If the function is called as a statement (see §4.4.5), then its return list is adjusted to 0, thus discarding all returned values. If the function is called in a place that needs a single value (syntactically denoted by the non-terminal *exp1*), then its return list is adjusted to 1, thus discarding all returned values but the first one. If the function is called in a place that can hold many values (syntactically denoted by the non-terminal *exp*), then no adjustment is made. The only places that can hold many values is the last (or the only) expression in an assignment, in an argument list, or in the **return** statement. 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
a,b,c = f()         -- f() is adjusted to 3
return f()          -- returns all values returned by f()
return x,y,f()      -- returns a, b, and all values returned by f()
```

4.5.9 Function Definitions

The syntax for function definition is

$$\begin{array}{rcl} \mathit{function} & \to & \textbf{function}\ \text{`('}\ [\mathit{parlist1}]\ \text{`)'}\ \mathit{block}\ \textbf{end} \\ \mathit{stat} & \to & \textbf{function}\ \mathit{funcname}\ \text{`('}\ [\mathit{parlist1}]\ \text{`)'}\ \mathit{block}\ \textbf{end} \\ \mathit{funcname} & \to & \mathit{name}\ \mid \mathit{name}\ \text{`.'}\ \mathit{name}\ \mid \mathit{name}\ \text{`:'}\ \mathit{name} \end{array}$$

The statement

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

is just syntactic sugar for

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

and the statement

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

is syntactic sugar for

```
v.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, its upvalues are fixed (see §4.6), and 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 have different upvalues.

Parameters act as local variables, initialized with the argument values:

parlist1 → '`...`'
parlist1 → *name* {'`,`' *name*} ['`,`' '`...`']

When a function is called, the list of arguments is adjusted to the length of the list of parameters (see §4.3), unless the function is a *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` whose value is the number of extra arguments, and 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 §4.4.3). If control reaches the end of a function without encountering a **return** statement, then the function returns with no results.

The syntax

funcname → *name* '`:`' *name*

is used for defining *methods*, that is, functions that have an implicit extra parameter `self`.

The statement

```
function v:f (...) ... end
```

is just syntactic sugar for

```
v.f = function (self, ...) ... end
```

Note that the function gets an extra formal parameter called `self`.

4.6 Visibility and Upvalues

A function body may refer to its own local variables (which include its parameters) and to global variables, as long as they are not *shadowed* by local variables with the same name from enclosing functions. A function *cannot* access a local variable from an enclosing function, since such variables may no longer exist when the function is called. However, a function may access the *value* of a local variable from an enclosing function, using *upvalues*, whose syntax is

upvalue → '%' *name*

An upvalue is somewhat similar to a variable expression, but whose value is *frozen* when the function wherein it appears is instantiated. The name used in an upvalue may be the name of any variable visible at the point where the function is defined, that is, global variables and local variables from the *immediately enclosing* function. Note that when the upvalue is a table, only the *reference* to that table (which is the value of the upvalue) is frozen; the table contents can be changed at will. Using table values as upvalues is a technique for having writable but private state attached to functions.

Here are some examples:

```
a,b,c = 1,2,3   -- global variables
local d
function f (x)
  local b = {}  -- x and b are local to f; b shadows the global b
  local g = function (a)
    local y     -- a and y are local to g
    p = a       -- OK, access local 'a'
    p = c       -- OK, access global 'c'
    p = b       -- ERROR: cannot access a variable in outer scope
    p = %b      -- OK, access frozen value of 'b' (local to 'f')
    %b = 3      -- ERROR: cannot change an upvalue
    %b.x = 3    -- OK, change the table contents
    p = %c      -- OK, access frozen value of global 'c'
    p = %y      -- ERROR: 'y' is not visible where 'g' is defined
    p = %d      -- ERROR: 'd' is not visible where 'g' is defined
  end           -- g
end             -- f
```

4.7 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. Whenever an error occurs during Lua compilation or execution, the function `_ERRORMESSAGE` is called (provided it is different from **nil**), and then the corresponding function from the library (`lua_dofile`, `lua_dostring`, `lua_dobuffer`, or `lua_call`) is terminated, returning an error condition.

Memory allocation errors are an exception to the previous rule. When memory allocation fails, Lua may not be able to execute the `_ERRORMESSAGE` function. So, for this kind of error, Lua does not call the `_ERRORMESSAGE` function; instead, the corresponding function from the library returns immediately with a special error code (`LUA_ERRMEM`). This and other error codes are defined in `lua.h`; §5.8.

The only argument to `_ERRORMESSAGE` is a string describing the error. The default definition for this function calls `_ALERT`, which prints the message to `stderr` (see §6.1). The standard I/O library redefines `_ERRORMESSAGE` and uses the debug facilities (see §7) to print some extra information, such as a call stack traceback.

Lua code can explicitly generate an error by calling the function `error` (see §6.1). Lua code can "catch" an error using the function `call` (see §6.1).

4.8 Tag Methods

Lua provides a powerful mechanism to extend its semantics, called *tag methods*. A tag method is a programmer-defined function that is called at specific key points during the execution of a Lua program, allowing the programmer to change the standard Lua behavior at these points. Each of these points is called an *event*.

The tag method called for any specific event is selected according to the tag of the values involved in the event (see §3). The function `settagmethod` changes the tag method associated with a given pair *(tag, event)*. Its first parameter is the tag, the second parameter is the event name (a string; see below), and the third parameter is the new method (a function), or **nil** to restore the default behavior for the pair. The `settagmethod` function returns the previous tag method for that pair. A companion function `gettagmethod` receives a tag and an event name and returns the current method associated with the pair.

Tag methods are called in the following events, identified by the given names. The semantics of tag methods is better explained by a Lua function describing the behavior of the interpreter at each event. This function not only shows when a tag method is called, but also its arguments, its results, and the default behavior. The code shown here 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`, `call`, etc.) are described in §6.1.

"add": called when a + operation is applied to non-numerical operands.

The function `getbinmethod` below defines how Lua chooses a tag method for a binary operation. First, Lua tries the first operand. If its tag does not define a tag method for the operation, then Lua tries the second operand. If it also fails, then it gets a tag method from tag 0.

```
function getbinmethod (op1, op2, event)
  return gettagmethod(tag(op1), event) or
         gettagmethod(tag(op2), event) or
         gettagmethod(0, event)
end
```

Using this function, the tag method for the "add" event 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 tm = getbinmethod(op1, op2, "add")
    if tm then
```

```
          -- call the method with both operands and an extra
          -- argument with the event name
          return tm(op1, op2, "add")
        else  -- no tag method available: default behavior
          error("unexpected type at arithmetic operation")
        end
      end
    end
```

"sub": called when a `-` operation is applied to non-numerical operands. Behavior similar to the "add" event.

"mul": called when a `*` operation is applied to non-numerical operands. Behavior similar to the "add" event.

"div": called when a `/` operation is applied to non-numerical operands. Behavior similar to the "add" event.

"pow": called when a `^` operation (exponentiation) is applied, even for numerical operands.

```
    function pow_event (op1, op2)
      local tm = getbinmethod(op1, op2, "pow")
      if tm then
        -- call the method with both operands and an extra
        -- argument with the event name
        return tm(op1, op2, "pow")
      else  -- no tag method available: default behavior
        error("unexpected type at arithmetic operation")
      end
    end
```

"unm": called when a unary `-` operation is applied to a non-numerical operand.

```
    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 tag method from the operand;
        --  if it does not have one, try a "global" one (tag 0)
        local tm = gettagmethod(tag(op), "unm") or
                   gettagmethod(0, "unm")
        if tm then
          -- call the method with the operand, nil, and an extra
          -- argument with the event name
          return tm(op, nil, "unm")
        else  -- no tag method available: default behavior
          error("unexpected type at arithmetic operation")
        end
    end
  end
```

"lt": called when an order operation is applied to non-numerical or non-string operands. It corresponds to the < operator.

```
  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 tm = getbinmethod(op1, op2, "lt")
      if tm then
        return tm(op1, op2, "lt")
      else
        error("unexpected type at comparison");
      end
    end
  end
```

The other order operators use this tag method according to the usual equivalences:

```
  a>b    <=>  b<a
  a<=b   <=>  not (b<a)
  a>=b   <=>  not (a<b)
```

"concat": called when a concatenation is applied to non-string operands.

```
  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 tm = getbinmethod(op1, op2, "concat")
      if tm then
        return tm(op1, op2, "concat")
      else
        error("unexpected type for concatenation")
      end
    end
  end
```

"index": called when Lua tries to retrieve the value of an index not present in a table. See the "gettable" event for its semantics.

"getglobal": called whenever Lua needs the value of a global variable. This method can only be set for **nil** and for tags created by `newtag`. Note that the tag is that of the *current value* of the global variable.

```
function getglobal (varname)
  -- access the table of globals
  local value = rawget(globals(), varname)
  local tm = gettagmethod(tag(value), "getglobal")
  if not tm then
    return value
  else
    return tm(varname, value)
  end
end
```

The function `getglobal` is defined in the basic library (see §6.1).

"setglobal": called whenever Lua assigns to a global variable. This method cannot be set for numbers, strings, and tables and userdata with the default tag.

```
function setglobal (varname, newvalue)
  local oldvalue = rawget(globals(), varname)
  local tm = gettagmethod(tag(oldvalue), "setglobal")
  if not tm then
    rawset(globals(), varname, newvalue)
  else
    tm(varname, oldvalue, newvalue)
  end
end
```

The function `setglobal` is defined in the basic library (see §6.1).

"gettable": called whenever Lua accesses an indexed variable. This method cannot be set for tables with the default tag.

```
function gettable_event (table, index)
  local tm = gettagmethod(tag(table), "gettable")
  if tm then
    return tm(table, index)
  elseif type(table) ~= "table" then
    error("indexed expression not a table");
  else
    local v = rawget(table, index)
    tm = gettagmethod(tag(table), "index")
    if v == nil and tm then
      return tm(table, index)
    else
      return v
    end
  end
end
```

"settable": called when Lua assigns to an indexed variable. This method cannot be set for tables with the default tag.

```
function settable_event (table, index, value)
  local tm = gettagmethod(tag(table), "settable")
  if tm then
    tm(table, index, value)
  elseif type(table) ~= "table" then
    error("indexed expression not a table")
  else
    rawset(table, index, value)
  end
end
```

"function": called when Lua tries to call a non-function value.

```
function function_event (func, ...)
  if type(func) == "function" then
    return call(func, arg)
  else
    local tm = gettagmethod(tag(func), "function")
    if tm then
      for i=arg.n,1,-1 do
        arg[i+1] = arg[i]
      end
      arg.n = arg.n+1
      arg[1] = func
      return call(tm, arg)
    else
      error("call expression not a function")
    end
  end
end
```

"gc": called when Lua is "garbage collecting" a userdata. This tag method can be set only from C, and cannot be set for a userdata with the default tag. For each userdata to be collected, Lua does the equivalent of the following function:

```
function gc_event (obj)
  local tm = gettagmethod(tag(obj), "gc")
  if tm then
    tm(obj)
  end
end
```

In a garbage-collection cycle, the tag methods for userdata are called in *reverse* order of tag creation, that is, the first tag methods to be called are those associated with the last tag created in the program. Moreover, at the end of the cycle, Lua does the equivalent of the call `gc_event(nil)`.

5 The Application Program Interface

This section describes the 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`.

NOTE: 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, and so do not generate hidden side-effects.

5.1 States

The Lua library is fully reentrant: it does not have any global variables. The whole state of the Lua interpreter (global variables, stack, tag methods, etc.) is stored in a dynamically allocated structure of type `lua_State`; this state must be passed as the first argument to every function in the library (except `lua_open` below).

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

```
       lua_State *lua_open (int stacksize);
```

The sole argument to this function is the stack size for the interpreter. (Each function call needs one stack position for each argument, local variable, and temporary value, plus one position for book-keeping. The stack must also have some 20 extra positions available. For very small implementations, without recursive functions, a stack size of 100 should be enough.) If `stacksize` is zero, then a default size of 1024 is used.

To release a state created with `lua_open`, call

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

This function destroys all objects in the given Lua environment (calling the corresponding garbage-collection tag methods, if any) and frees all dynamic memory used by that state. Usually, you do not need to call this function, because all resources are naturally released when your program ends. On the other hand, long-running programs — like a daemon or a web server — might need to release states as soon as they are not needed, to avoid growing too big.

With the exception of `lua_open`, all functions in the Lua API need a state as their first argument.

5.2 The Stack and Indices

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

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, not 0 as in C); a negative index represents an *offset* from the top of the stack. More specifically, if the stack has n elements, index 1 represents the first element (that is, the first element pushed onto the stack), 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 lays 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

```
       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_stackspace (lua_State *L);
```

returns the number of stack positions still available. Whenever Lua calls C, it ensures that at least `LUA_MINSTACK` positions are still available. `LUA_MINSTACK` is defined in `lua.h` and is at least 16, and so you have to worry about stack space only when your code has loops pushing elements onto the stack.

Most query functions accept as indices any value inside the available stack space. Such indices are called *acceptable indices.* More formally, we can define an *acceptable index* as

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

Note that 0 is not an acceptable index.

5.3 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);
```

`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 API 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 on top of that position to fill in the gap. `lua_insert` moves the top element into the given position, shifting up the elements on top of that position to open space. These functions accept only valid indices. As an example, if the stack starts as `10 20 30 40 50` (from bottom to 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_settop(L, -3)      --> 30 10 20
lua_settop(L, 6)       --> 30 10 20 nil nil nil
```

5.4 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_tag         (lua_State *L, int index);
int        lua_isnil       (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);
```

These functions can be called with any acceptable index.

`lua_type` returns one of the following constants, according to the type of the given object: `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`. If the index is non-valid (that is, if that stack position is "empty"), then `lua_type` returns `LUA_TNONE`. These constants can be converted to strings with

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

where `t` is a type returned by `lua_type`. The strings returned by `lua_typename` are `"nil"`, `"number"`, `"string"`, `"table"`, `"function"`, `"userdata"`, and `"no value"`,

`lua_tag` returns the tag of a value, or `LUA_NOTAG` for a non-valid index.

The `lua_is*` functions return 1 if the object is compatible with the given type, and 0 otherwise. They always return 0 for a non-valid index. `lua_isnumber` accepts numbers and numerical strings, `lua_isstring` accepts strings and numbers (see §4.2), and `lua_isfunction` accepts both Lua functions and C functions. To distinguish between Lua functions and C functions, you should use `lua_iscfunction`. To distinguish between numbers and numerical strings, you can use `lua_type`.

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

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

These functions are equivalent to their counterparts in Lua. Specifically, `lua_lessthan` is equivalent to the `lt_event` described in §4.8. Both functions return 0 if any of the indices are non-valid.

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

```
double         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);
```

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_tonumber` converts the value at the given index to a floating-point number. This value must be a number or a string convertible to number (see §4.2); otherwise, `lua_tonumber` returns 0.

`lua_tostring` converts a Lua value to a string (`const char*`). This value must be a string or a number; otherwise, the function returns `NULL`. This function returns a pointer to a string inside the Lua environment. Those strings always have a zero (`'\0'`) after their last character (as in C), but may contain other zeros in their body. If you do not know whether a string may contain zeros, you should 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 respective value is removed from the stack.

`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 §5.13.

`lua_touserdata` converts a value to `void*`. This value must have type *userdata*; otherwise, `lua_touserdata` returns `NULL`.

5.5 Pushing values onto the Stack

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

```
void lua_pushnumber    (lua_State *L, double n);
void lua_pushlstring   (lua_State *L, const char *s, size_t len);
void lua_pushstring    (lua_State *L, const char *s);
void lua_pushusertag   (lua_State *L, void *u, int tag);
void lua_pushnil       (lua_State *L);
void lua_pushcfunction (lua_State *L, lua_CFunction f);
```

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.

5.6 Garbage Collection

Lua uses two numbers to control its garbage collection. One number counts how many bytes of dynamic memory Lua is using, and the other is a threshold. (This internal byte counter kept by Lua is not completely acurate; it is just a lower bound, usually within 10% of the correct value.) When the number of bytes crosses the threshold, Lua runs a garbage-collection cycle, which reclaims the memory of all “dead” objects (that is, objects no longer accessible from Lua). The byte counter is corrected, and then the threshold is reset to twice the value of the byte counter.

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 smaller than the byte counter, then Lua immediately runs the garbage collector; after the collection, a new threshold is set according to the previous rule.

If you want to change the adaptative behavior of the garbage collector, you can use the garbage-collection tag method for **nil** to set your own threshold (the tag method is called after Lua resets the threshold).

5.7 Userdata and Tags

Because userdata are objects, the function `lua_pushusertag` may create a new userdata. If Lua has a userdata with the given value (`void*`) and tag, then that userdata is pushed. Otherwise, a new userdata is created, with the given value and tag. If this function is called with `tag` equal to `LUA_ANYTAG`, then Lua will try to find any userdata with the given value, regardless of its tag. If there is no userdata with that value, then a new one is created, with tag equal to 0.

Userdata can have different tags, whose semantics are only known to the host program. Tags are created with the function

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

The function `lua_settag` changes the tag of the object on top of the stack (without popping it):

```
void lua_settag (lua_State *L, int tag);
```

The object must be a userdata or a table; the given `tag` must be a value created with `lua_newtag`.

5.8 Executing Lua Code

A host program can execute Lua chunks written in a file or in a string by using the following functions:

```
int lua_dofile   (lua_State *L, const char *filename);
int lua_dostring (lua_State *L, const char *string);
int lua_dobuffer (lua_State *L, const char *buff,
                  size_t size, const char *name);
```

These functions return 0 in case of success, or one of the following error codes if they fail:

- `LUA_ERRRUN` — error while running the chunk.
- `LUA_ERRSYNTAX` — syntax error during pre-compilation.
- `LUA_ERRMEM` — memory allocation error. For such errors, Lua does not call `_ERRORMESSAGE` (see §4.7).
- `LUA_ERRERR` — error while running `_ERRORMESSAGE`. For such errors, Lua does not call `_ERRORMESSAGE` again, to avoid loops.
- `LUA_ERRFILE` — error opening the file (only for `lua_dofile`). In this case, you may want to check `errno`, call `strerror`, or call `perror` to tell the user what went wrong.

These constants are defined in `lua.h`.

When called with argument `NULL`, `lua_dofile` executes the `stdin` stream. `lua_dofile` and `lua_dobuffer` are both able to execute pre-compiled chunks. They automatically detect whether the chunk is text or binary, and load it accordingly (see program `luac`). `lua_dostring` executes only source code, given in textual form.

The third parameter to lua_dobuffer is the "name of the chunk", which is used in error messages and debug information. If name is NULL, then Lua gives a default name to the chunk.

These functions push onto the stack any values eventually returned by the chunk. A chunk may return any number of values; Lua takes care that these values fit into the stack space, but after the call the responsibility is back to you. If you need to push other elements after calling any of these functions, and you want to "play safe", you must either check the stack space with lua_stackspace or remove the returned elements from the stack (if you do not need them). For instance, the following code loads a chunk in a file and discards all results returned by this chunk, leaving the stack as it was before the call:

```
{
 int oldtop = lua_gettop(L);
 lua_dofile(L, filename);
 lua_settop(L, oldtop);
}
```

5.9 Manipulating Global Variables in Lua

To read the value of a global Lua variable, you call

```
void lua_getglobal (lua_State *L, const char *varname);
```

which pushes onto the stack the value of the given variable. As in Lua, this function may trigger a tag method for the "getglobal" event (see §4.8). To read the real value of a global variable, without invoking any tag method, use lua_rawget over the table of globals (see below).

To store a value in a global variable, you call

```
void lua_setglobal (lua_State *L, const char *varname);
```

which pops from the stack the value to be stored in the given variable. As in Lua, this function may trigger a tag method for the "setglobal" event (see §4.8). To set the real value of a global variable, without invoking any tag method, use lua_rawset over the table of globals (see below).

All global variables are kept in an ordinary Lua table. You can get this table calling

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

which pushes the current table of globals onto the stack. To set another table as the table of globals, you call

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

The table to be used is popped from the stack.

5.10 Manipulating Tables in Lua

Lua tables can also be manipulated through the API.

To read the value of in a table, the table must reside somewhere in the stack. With this set, you call

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

where `index` refers to the table. `lua_gettable` pops a key from the stack, and returns (on the stack) the contents of the table at that key. As in Lua, this operation may trigger a tag method for the "gettable" event. To get the real value of any table key, without invoking any tag method, 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 the value onto the stack (in this order), and then call

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

where `index` refers to the table. `lua_settable` pops from the stack both the key and the value. As in Lua, this operation may trigger a tag method for the "settable" event. To set the real value of any table index, without invoking any tag method, use the *raw* version:

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

Finally, the function

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

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

5.11 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);
int  lua_getn    (lua_State *L, int index);
```

`lua_rawgeti` gets 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.

`lua_getn` returns the number of elements in the table at stack position `index`. This number is the value of the table field `n`, if it has a numeric value, or the largest numerical index with a non-nil value in the table.

5.12 Calling Lua 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 (see §5.5) in *direct order*, that is, the first argument is pushed first. Finally, the function is called using

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

This function returns the same error codes as `lua_dostring` and friends (see §5.8). If you want to propagate the error, instead of returning an error code, use

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

In both functions, 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 (see §4.3) to nresults, unless nresults is LUA_MULTRET. In that case, *all* results from the function are pushed. The function results are pushed 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 the Lua code:

```
       a,b = f("how", t.x, 4)
```

Here it is in C:

```
   lua_getglobal(L, "t");                          /* global 't' (for later use) */
   lua_getglobal(L, "f");                               /* function to be called */
   lua_pushstring(L, "how");                                     /* 1st argument */
   lua_pushstring(L, "x");                                /* push the string 'x' */
   lua_gettable(L, -4);                          /* push result of t.x (2nd arg) */
   lua_pushnumber(L, 4);                                         /* 3rd argument */
   lua_call(L, 3, 2);       /* call function with 3 arguments and 2 results */
   lua_setglobal(L, "b");                             /* set global variable 'b' */
   lua_setglobal(L, "a");                             /* set global variable 'a' */
   lua_pop(L, 1);                                   /* remove 't' from the stack */
```

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

Some special Lua functions have their own C interfaces. The host program can generate a Lua error calling the function

```
       void lua_error (lua_State *L, const char *message);
```

This function never returns. If lua_error is called from a C function that has been called from Lua, then the corresponding Lua execution terminates, as if an error had occurred inside Lua code. Otherwise, the whole host program terminates with a call to exit(EXIT_FAILURE). Before terminating execution, the message is passed to the error handler function, _ERRORMESSAGE (see §4.7). If message is NULL, then _ERRORMESSAGE is not called.

Tag methods can be changed with

```
       void lua_settagmethod (lua_State *L, int tag, const char *event);
```

The second parameter is the tag, and the third is the event name (see §4.8); the new method is popped from the stack. To get the current value of a tag method, use the function

```
       void lua_gettagmethod (lua_State *L, int tag, const char *event);
```

It is also possible to copy all tag methods from one tag to another:

```
       int lua_copytagmethods (lua_State *L, int tagto, int tagfrom);
```

This function returns tagto.

You can traverse a table with the function

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

where index refers 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 the function returns 0 (and pushes nothing). 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 ‘index’ for next iteration */
}
```

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; n must be at least 2. Concatenation is done following the usual semantics of Lua (see §4.5.5).

5.13 Defining C Functions

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

```
#define lua_register(L, n, f) (lua_pushcfunction(L, f), lua_setglobal(L, n))
/* const char *n;   */
/* lua_CFunction f; */
```

which receives the name the function will have in Lua, and a pointer to the function. This pointer must have type lua_CFunction, which is defined as

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

that is, a pointer to a function with integer result and a single argument, a Lua environment.

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 the stack, in direct order (the first argument is pushed first). 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. 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 */
  double sum = 0;
  int i;
  for (i = 1; i <= n; i++) {
    if (!lua_isnumber(L, i))
      lua_error(L, "incorrect argument to function ‘average’");
    sum += lua_tonumber(L, i);
    }
    lua_pushnumber(L, sum/n);        /* first result */
    lua_pushnumber(L, sum);         /* second result */
    return 2;                   /* number of results */
  }
```

This function may be registered in Lua as '`average`' by calling

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

When a C function is created, it is possible to associate some *upvalues* to it (see §4.6), thus creating a *C closure*; these values are passed to the function whenever it is called, as ordinary arguments. To associate upvalues to a C function, first these values should be pushed onto the stack. 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 upvalues should be associated with the function (these upvalues are popped 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, these upvalues are inserted as the *last* arguments to the function, after the actual arguments provided in the call. This makes it easy to get the upvalues without knowing how many arguments the function received (recall that functions in Lua can receive any number of arguments): The i-th upvalue is in the stack at index $i - (n + 1)$, where n is the number of upvalues.

For more examples of C functions and closures, see files `lbaselib.c`, `liolib.c`, `lmathlib.c`, and `lstrlib.c` in the official Lua distribution.

5.14 References to Lua Objects

If the C code needs to keep a Lua value outside the life span of a C function, then it must create a *reference* to the value. The functions to manipulate references are the following:

```
  int   lua_ref    (lua_State *L, int lock);
  int   lua_getref (lua_State *L, int ref);
  void  lua_unref  (lua_State *L, int ref);
```

`lua_ref` pops a value from the stack, creates a reference to it, and returns this reference. For a **nil** value, the reference is always `LUA_REFNIL`. (`lua.h` also defines a constant `LUA_NOREF` that is different from any valid reference.) If `lock` is not zero, then the object is *locked*: this means the object will not be garbage collected. *Unlocked references may be garbage collected.*

Whenever the referenced object is needed in C, a call to `lua_getref` pushes that object onto the stack; if the object has been collected, `lua_getref` returns 0 (and does not push anything).

When a reference is no longer needed, it should be released with a call to `lua_unref`.

Registry

When Lua starts, it registers a table at position `LUA_REFREGISTRY`. It can be accessed through the macro

```
 #define lua_getregistry(L)      lua_getref(L, LUA_REFREGISTRY)
```

This table can be used by C libraries as a general registry mechanism. Any C library can store data into this table, as long as it chooses a key different from other libraries.

6 Standard Libraries

The standard libraries provide useful functions that are implemented directly through the standard API. Therefore, they are not necessary to the language, and are provided as separate C modules. Currently, Lua has the following standard libraries:

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

To have access to these libraries, the C host program must call the functions `lua_baselibopen`, `lua_strlibopen`, `lua_mathlibopen`, and `lua_iolibopen`, which are declared in `lualib.h`.

6.1 Basic Functions

The basic library provides some core functions to Lua. Therefore, if you do not include this library in your application, you should check carefully whether you need to provide some alternative implementation for some facilities. (For instance, without function `_ERRORMESSAGE`, Lua is unable to show error messages.)

• `_ALERT (message)`

Prints its only string argument to `stderr`. All error messages in Lua are printed through the function stored in the `_ALERT` global variable (see §4.7). Therefore, a program may assign another function to this variable to change the way such messages are shown (for instance, for systems without `stderr`).

• `assert (v [, message])`

Issues an *"assertion failed!"* error when its argument `v` is **nil**. This function is equivalent to the following Lua function:

```
function assert (v, m)
  if not v then
    m = m or ""
    error("assertion failed!  " .. m)
  end
end
```

• `call (func, arg [, mode [, errhandler]])`

Calls function `func` with the arguments given by the table `arg`. The call is equivalent to

```
func(arg[1], arg[2], ..., arg[n])
```

where `n` is the result of `getn(arg)` (see §6.1). All results from `func` are simply returned by `call`.

By default, if an error occurs during the call to `func`, the error is propagated. If the string `mode` contains `"x"`, then the call is *protected*. In this mode, function `call` does not propagate an error, regardless of what happens during the call. Instead, it returns **nil** to signal the error (besides calling the appropriated error handler).

If `errhandler` is provided, the error function `_ERRORMESSAGE` is temporarily set to `errhandler`, while `func` runs. In particular, if `errhandler` is **nil**, no error messages will be issued during the execution of the called function.

• `collectgarbage ([limit])`

Sets the garbage-collection threshold for 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 §5.6). If `limit` is absent, it defaults to zero (thus forcing a garbage-collection cycle).

• `copytagmethods (tagto, tagfrom)`

Copies all tag methods from one tag to another; returns `tagto`.

• `dofile (filename)`

Receives a file name, opens the named file, and executes its contents as a Lua chunk, or as pre-compiled chunks. When called without arguments, `dofile` executes the contents of the standard input (`stdin`). If there is any error executing the file, then `dofile` returns **nil**. Otherwise, it returns the values returned by the chunk, or a non-**nil** value if the chunk returns no values. It issues an error when called with a non-string argument.

• `dostring (string [, chunkname])`

Executes a given string as a Lua chunk. If there is any error executing the string, then `dostring` returns **nil**. Otherwise, it returns the values returned by the chunk, or a non-**nil** value if the chunk returns no values. The optional parameter `chunkname` is the "name of the chunk", used in error messages and debug information.

• `error (message)`

Calls the error handler (see §4.7) and then terminates the last protected function called (in C: `lua_dofile`, `lua_dostring`, `lua_dobuffer`, or `lua_callfunction`; in Lua: `dofile`, `dostring`, or `call` in protected mode). If `message` is **nil**, then the error handler is not called. Function `error` never returns.

• `foreach (table, func)`

Executes the given `func` over all elements of `table`. For each element, the function is called with the index and respective value as arguments. If the function returns any non-**nil** value, then the loop is broken, and this value is returned as the final value of `foreach`. This function could be defined in Lua:

```
function foreach (t, f)
  for i, v in t do
    local res = f(i, v)
    if res then return res end
  end
end
```

The behavior of `foreach` is *undefined* if you change the table `t` during the traversal.

- `foreachi (table, func)`

Executes the given `func` over the numerical indices of `table`. For each index, the function is called with the index and respective value as arguments. Indices are visited in sequential order, from 1 to `n`, where `n` is the result of `getn(table)` (see §6.1). If the function returns any non-**nil** value, then the loop is broken, and this value is returned as the final value of `foreachi`. This function could be defined in Lua:

```
function foreachi (t, f)
  for i=1,getn(t) do
    local res = f(i, t[i])
    if res then return res end
  end
end
```

- `getglobal (name)`

Gets the value of a global variable, or calls a tag method for "getglobal". Its full semantics is explained in §4.8. The string `name` does not need to be a syntactically valid variable name.

- `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, the "size" is the largest numerical index with a non-nil value in the table. This function could be defined in Lua:

```
function getn (t)
  if type(t.n) == "number" then return t.n end
  local max = 0
  for i, _ in t do
    if type(i) == "number" and i>max then max=i end
  end
  return max
end
```

- `gettagmethod (tag, event)`

Returns the current tag method for a given pair *(tag, event)*. This function cannot be used to get a tag method for the "gc" event. (Such tag methods can only be manipulated by C code.)

• `globals ([table])`

Returns the current table of globals. If the argument `table` is given, then it also sets this table as the table of globals.

• `newtag ()`

Returns a new tag.

• `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; semantically, 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 function `foreachi`).

The behavior of `next` is *undefined* if you change the table during the traversal.

• `print (e1, e2, ...)`

Receives any number of arguments, and prints their values using the strings returned by `tostring`. This function is not intended for formatted output, but only as a quick way to show a value, for instance for debugging. See §6.4 for functions for formatted output.

• `rawget (table, index)`

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

• `rawset (table, index, value)`

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

• `setglobal (name, value)`

Sets the named global variable to the given value, or calls a tag method for "setglobal". Its full semantics is explained in §4.8. The string `name` does not need to be a syntactically valid variable name.

• `settag (t, tag)`

Sets the tag of a given table (see §3). `tag` must be a value created with `newtag` (see §6.1). `settag` returns the value of its first argument (the table). For the safety of host programs, it is impossible to change the tag of a userdata from Lua.

• `settagmethod (tag, event, newmethod)`

Sets a new tag method to the given pair *(tag, event)* and returns the old method. If `newmethod` is **nil**, then `settagmethod` restores the default behavior for the given event. This function cannot be used to set a tag method for the "gc" event. (Such tag methods can only be manipulated by C code.)

• `sort (table [, comp])`

Sorts table elements in a given order, *in-place*, from `table[1]` to `table[n]`, where `n` is the result of `getn(table)` (see §6.1). If `comp` is given, then it must be a function that receives two table elements, and returns true (that is, a value different from **nil**) 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).

• `tag (v)`

Allows Lua programs to test the tag of a value (see §3). It receives one argument, and returns its tag (a number).

• `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' (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 §4.2). 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 function `format`.

• `tinsert (table [, pos] , value)`

Inserts element `value` at table position `pos`, shifting other elements to open space, if necessary. The default value for `pos` is `n+1`, where `n` is the result of `getn(table)` (see §6.1), so that a call `tinsert(t,x)` inserts `x` at the end of table `t`. This function also sets or increments the field `n` of the table to `n+1`. This function is equivalent to the following Lua function, except that the table accesses are all *raw* (that is, without tag methods):

```
function tinsert (t, ...)
  local pos, value
  local n = getn(t)
  if arg.n == 1 then
    pos, value = n+1, arg[1]
    else
      pos, value = arg[1], arg[2]
    end
    t.n = n+1;
    for i=n,pos,-1 do
      t[i+1] = t[i]
    end
    t[pos] = value
  end
```

• `tremove (table [, pos])`

Removes from `table` the element at position `pos`, shifting 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 result of `getn(table)` (see §6.1), so that a call `tremove(t)` removes the last element of table `t`. This function also sets or decrements the field `n` of the table to `n-1`.

This function is equivalent to the following Lua function, except that the table accesses are all *raw* (that is, without tag methods):

```
  function tremove (t, pos)
    local n = getn(t)
    if n<=0 then return end
    pos = pos or n
    local value = t[pos]
    for i=pos,n-1 do
      t[i] = t[i+1]
    end
    t[n] = nil
    t.n = n-1
    return value
  end
```

• `type (v)`

Allows Lua programs to test the type of a value. It receives one argument, and returns its type, coded as a string. The possible results of this function are `"nil"` (a string, not the value **nil**), `"number"`, `"string"`, `"table"`, `"function"`, and `"userdata"`.

6.2 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). Also, indices are allowed to be negative and are intepreted as indexing backwards, from the end of the string. Thus, the last character is at position -1, and so on.

• `strbyte (s [, i])`

Returns the internal numerical code of the i-th character of `s`. If `i` is absent, then it is assumed to be 1. `i` may be negative.

NOTE: Numerical codes are not necessarily portable across platforms.

• `strchar (i1, i2, ...)`

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

NOTE: Numerical codes are not necessarily portable across platforms.

• `strfind (s, pattern [, init [, plain]])`

Looks for the first *match* of `pattern` in `s`. If it finds one, then `strfind` returns the indices of `s` where this occurrence starts and ends; otherwise, it returns **nil**. If the pattern specifies captures (see `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 1 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.

• `strlen (s)`

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

• `strlower (s)`

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

• `strrep (s, n)`

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

• `strsub (s, i [, j])`

Returns another string, which is a substring of `s`, starting at `i` and running 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 `strsub(s,1,j)` returns a prefix of `s` with length `j`, and the call `strsub(s, -i)` returns a suffix of `s` with length `i`.

• `strupper (s)`

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

• `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, returns, and backslashes in the string are correctly escaped when written. For instance, the call

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

will produce the string:

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

Conversions can be applied to the *n*-th argument in the argument list, rather than the next unused argument. In this case, the conversion character `%` is replaced by the sequence `%d$`, where `d` is a decimal digit in the range [1,9], giving the position of the argument in the argument list. For instance, the call `format("%2$d -> %1$03d", 1, 34)` will result in `"34 -> 001"`. The same argument can be used in more than one conversion.

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"`.

NOTE: Neither the format string nor the string values to be formatted with `%s` can contain embedded zeros. `%q` handles string values with embedded zeros.

• `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.

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order (see below). 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 last, optional 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 = gsub("hello world", "(%w+)", "%1 %1")
--> x="hello hello world world"

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

x = gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")
```

```
--> x="world hello Lua from"

x = gsub("home = $HOME, user = $USER", "%$(%w+)", getenv)
--> x="home = /home/roberto, user = roberto"  (for instance)

x = gsub("4+5 = $return 4+5$", "%$(.-)%$", dostring)
--> x="4+5 = 9"

local t = {name="lua", version="4.0"}
x = gsub("$name - $version", "%$(%w+)", function (v) return %t[v] end)
--> x="lua - 4.0"

t = {n=0}
gsub("first second word", "(%w+)", function (w) tinsert(%t, w) end)
--> t={"first", "second", "word"; n=3}
```

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 any 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 lower case letters.

`%p` — represents all punctuation characters.

`%s` — represents all space characters.

`%u` — represents all upper case 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. We recommend that any punctuation character (even the non magic) should be preceded by a `%` when used to represent itself in a pattern.

`[char-set]` — represents the class which is the union of all characters in `char-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 a char-set. All other characters in char-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 lower case letters plus the `-` character.

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

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

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

The definitions of letter, space, etc. 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 sub-string 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 sub-strings 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.

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

6.3 Mathematical Functions

This library is an interface to some functions of the standard C math library. In addition, it registers a tag method for the binary operator `^` that returns x^y when applied to numbers `x^y`.

The library provides the following functions:

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

plus a global variable `PI`. Most of them are only interfaces to the homonymous functions in the C library, except that, for the trigonometric functions, all angles are expressed in *degrees*, not radians. The functions `deg` and `rad` can be used to convert between radians and degrees.

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

The functions `random` and `randomseed` are interfaces to the simple random generator functions `rand` and `srand`, provided by ANSI C. (No guarantees can be given for their statistical properties.) The function `random`, when called without arguments, returns a pseudo-random real number in the range $[0,1)$. When called with a number n, `random` returns a pseudo-random integer in the range $[1,n]$. When called with two arguments, l and u, `random` returns a pseudo-random integer in the range $[l,u]$.

6.4 I/O Facilities

All input and output operations in Lua are done, by default, over two *file handles*, one for reading and one for writing. These handles are stored in two Lua global variables, called `_INPUT` and `_OUTPUT`. The global variables `_STDIN`, `_STDOUT`, and `_STDERR` are initialized with file descriptors for `stdin`, `stdout`, and `stderr`. Initially, `_INPUT=_STDIN` and `_OUTPUT=_STDOUT`.

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

Unless otherwise stated, all I/O functions return **nil** on failure and some value different from **nil** on success.

• `openfile (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 a string describing the error. This function does not modify either `_INPUT` or `_OUTPUT`.

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

"r" read mode;

"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 exactlty what is used in the standard C function `fopen`.

• `closefile (handle)`

This function closes the given file. It does not modify either `_INPUT` or `_OUTPUT`.

• `readfrom (filename)`

This function may be called in two ways. When called with a file name, it opens the named file, sets its handle as the value of `_INPUT`, and returns this value. It does not close the current input file. When called without parameters, it closes the `_INPUT` file, and restores `stdin` as the value of `_INPUT`. If this function fails, it returns **nil**, plus a string describing the error.

NOTE: If `filename` starts with a `|`, then a piped input is opened, via function `popen`. Not all systems implement pipes. Moreover, the number of files that can be open at the same time is usually limited and depends on the system.

• `writeto (filename)`

This function may be called in two ways. When called with a file name, it opens the named file, sets its handle as the value of `_OUTPUT`, and returns this value. It does not close the current output file. Note that, if the file already exists, then it will be *completely erased* with this operation. When called without parameters, this function closes the `_OUTPUT` file, and restores `stdout` as the value of `_OUTPUT`. If this function fails, it returns **nil**, plus a string describing the error.

NOTE: If `filename` starts with a `|`, then a piped input is opened, via function `popen`. Not all systems implement pipes. Moreover, the number of files that can be open at the same time is usually limited and depends on the system.

• `appendto (filename)`

Opens a file named `filename` and sets it as the value of `_OUTPUT`. Unlike the `writeto` operation, this function does not erase any previous contents of the file; instead, anything written to the file is appended to its end. If this function fails, it returns **nil**, plus a string describing the error.

• `remove (filename)`

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

• `rename (name1, name2)`

Renames file named `name1` to `name2`. If this function fails, it returns **nil**, plus a string describing the error.

• `flush ([filehandle])`

Saves any written data to the given file. If `filehandle` is not specified, then `flush` flushes all open files. If this function fails, it returns **nil**, plus a string describing the error.

• `seek (filehandle [, whence] [, offset])`

Sets and gets the file position, measured in bytes 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 the call 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 `seek(file)` returns the current file position, without changing it; the call `seek(file, "set")` sets the position to the beginning of the file (and returns 0); and the call `seek(file, "end")` sets the position to the end of the file, and returns its size.

• `tmpname ()`

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

• `read ([filehandle,] format1, ...)`

Reads file `_INPUT`, or `filehandle` if this argument is given, 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 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.

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

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

"*w" reads the next word (maximal sequence of non–white-space characters), skipping spaces if necessary, or **nil** on end of file.

number reads a string with up to that number of characters, or **nil** on end of file.

• `write ([filehandle, ] value1, ...)`

Writes the value of each of its arguments to file `_OUTPUT`, or to `filehandle` if this argument is given. The arguments must be strings or numbers. To write other values, use `tostring` or `format` before `write`. If this function fails, it returns **nil**, plus a string describing the error.

6.5 System Facilities

• `clock ()`

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

• `date ([format])`

Returns a string containing date and time formatted according to the given string `format`, following the same rules of the ANSI C function `strftime`. When called without arguments, it returns a reasonable date and time representation that depends on the host system and on the current locale.

• `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.

• `exit ([code])`

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

• `getenv (varname)`

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

• `setlocale (locale [, category])`

This function is an interface to the ANSI C function `setlocale`. `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.

7 The Debug Interface

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

7.1 Stack and Function Information

The main function to get information about the interpreter stack is

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

It 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. Usually, `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 {
        const char *event;     /* "call", "return" */
        int currentline;       /* (l) */
        const char *name;      /* (n) */
        const char *namewhat;  /* (n) global, tag method, local, field */
      int nups;             /* (u) number of upvalues */
      int linedefined;      /* (S) */
      const char *what;     /* (S) "Lua" function, "C" function, Lua "main" */
      const char *source;   /* (S) */
      char short_src[LUA_IDSIZE]; /* (S) */

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

lua_getstack fills only the private part of this structure, for future use. To fill in 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 (e.g., an invalid option in what). Each character in the string what selects some fields of ar to be filled, as indicated by the letter in parentheses in the definition of lua_Debug: '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, it is not in the stack), you push the function 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_getglobal(L, "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, source is that string; if the function was defined in a file, 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, or "main" if this is the main part of a chunk.

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 many global variables, while others may be stored only in a table field. The lua_getinfo function checks whether the given function is a tag method or the value of a global variable. If the given function is a tag method, then name points to the event name. If the given function is the value of a global variable, then name points to the variable name. If the given function is neither a tag method nor a global variable, then name is set to NULL.

namewhat Explains the previous field. If the function is a global variable, `namewhat` is `"global"`; if the function is a tag method, `namewhat` is `"tag-method"`; otherwise `namewhat` is `""` (the empty string).

nups Number of upvalues of a function.

7.2 Manipulating Local Variables

For the manipulation of local variables, `luadebug.h` uses indices: The first parameter or local variable has index 1, and so on, until the last active local variable.

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, filled by a previous call to `lua_getstack` or given as argument to a hook (see §7.3). Function `lua_getlocal` gets the index of a local variable (`n`), pushes its value onto the stack, and returns its name. For `lua_setlocal`, you push the new value onto the stack, and the function assigns that value to the variable and returns its name. Both functions return `NULL` on failure; that happens if the index is greater than the number of active local variables.

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

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

7.3 Hooks

The Lua interpreter offers two hooks for debugging purposes: a *call* hook and a *line* hook. Both have the same type,

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

and you can set them with the following functions:

```
lua_Hook lua_setcallhook (lua_State *L, lua_Hook func);
lua_Hook lua_setlinehook (lua_State *L, lua_Hook func);
```

A hook is disabled when its value is `NULL`, which is the initial value of both hooks. The functions `lua_setcallhook` and `lua_setlinehook` set their corresponding hooks and return their previous values.

The call hook is called whenever the interpreter enters or leaves a function. The `event` field of `ar` has the strings `"call"` or `"return"`. This `ar` can then be used in calls to `lua_getinfo`, `lua_getlocal`, and `lua_setlocal` to get more information about the function and to manipulate its local variables.

The line hook is called every time the interpreter changes the line of code it is executing. The `event` field of `ar` has the string `"line"`, and the `currentline` field has the line number. Again, you can use this `ar` in other calls to the debug API.

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

7.4 The Reflexive Debug Interface

The library `ldblib` provides the functionality of the debug interface to Lua programs. If you want to use this library, your host application must open it, by calling `lua_dblibopen`.

You should exert great care when using this library. The functions provided here should be used exclusively for debugging and similar tasks (e.g., profiling). Please resist the temptation to use them as a usual programming tool. They are slow and violate some (otherwise) secure aspects of the language (e.g., privacy of local variables). As a general rule, if your program does not need this library, do not open it.

• `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 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 what to get. The default for `what` is to get all information available.

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

• `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 `getinfo` to check whether the level is valid.)

• `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.

• `setcallhook (hook)`

Sets the function `hook` as the call hook; this hook will be called every time the interpreter starts and exits the execution of a function. The only argument to the call hook is the event name (`"call"` or `"return"`). You can call `getinfo` with level 2 to get more information about the function being called or returning (level 0 is the `getinfo` function, and level 1 is the hook function). When called without arguments, this function turns off call hooks. `setcallhook` returns the old hook.

• `setlinehook (hook)`

Sets the function `hook` as the line hook; this hook will be called every time the interpreter changes the line of code it is executing. The only argument to the line hook is the line number the interpreter is about to execute. When called without arguments, this function turns off line hooks. `setlinehook` returns the old hook.

8 Lua Stand-alone

Although Lua has been designed as an extension language, to be embedded in a host C program, it is 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. This program can be called with any sequence of the following arguments:

`-sNUM` sets the stack size to `NUM` (if present, this must be the first option);

`-` executes `stdin` as a file;

`-c` calls `lua_close` after running all arguments;

`-e` *stat* executes string `stat`;

`-f filename` executes file `filename` with the remaining arguments in table `arg`;

`-i` enters interactive mode with prompt;

`-q` enters interactive mode without prompt;

`-v` prints version information;

`var=value` sets global `var` to string `"value"`;

`filename` executes file `filename`.

When called without arguments, `lua` behaves as `lua -v -i` when `stdin` is a terminal, and as `lua -` otherwise.

All arguments are handled in order, except `-c`. For instance, an invocation like

```
$ lua -i a=test prog.lua
```

will first interact with the user until an `EOF` in `stdin`, then will set `a` to `"test"`, and finally will run the file `prog.lua`. (Here, `$` is the shell prompt. Your prompt may be different.)

When the option `-f filename` is used, all remaining arguments in the command line are passed to the Lua program `filename` in a table called `arg`. In this table, the field `n` gets the index of the last argument, and the field 0 gets `"filename"`. For instance, in the call

```
$ lua a.lua -f b.lua t1 t3
```

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

```
arg = {"t1", "t3";  n = 2, [0] = "b.lua"}
```

and finally runs the file `b.lua`. The stand-alone interpreter also provides a `getargs` function that can be used to access *all* command line arguments. For instance, if you call Lua with the line

```
$ lua -c a b
```

then a call to `getargs` in `a` or `b` will return the table

```
{[0] = "lua", [1] = "-c", [2] = "a", [3] = "b", n = 3}
```

In interactive mode, a multi-line statement can be written finishing intermediate lines with a backslash ('\'). 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 _PROMPT='myprompt> ' -i
```

or in any Lua programs by assigning 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`, or `#!/usr/local/bin/lua -f` to get other arguments.

Acknowledgments

The authors would like to thank CENPES/PETROBRAS which, jointly with TeCGraf, used early versions of this system extensively and gave valuable comments. The authors would also like to thank Carlos Henrique Levy, who found the name of the game. Lua means "moon" in Portuguese.

Incompatibilities with Previous Versions

Lua 4.0 is a major revision of the language. We took a great care to avoid incompatibilities with the previous public versions of Lua, but some differences had to be introduced. Here is a list of all these incompatibilities.

Incompatibilities with version 3.2

Changes in the Language

- All pragmas (`$debug`, `$if`, ...) have been removed.
- **for**, **break**, and **in** are now reserved words.
- Garbage-collection tag methods for tables is now obsolete.
- There is now only one tag method for order operators.
- In nested function calls like `f(g(x))`, *all* return values from `g` are passed as arguments to `f`. This only happens when `g` is the last or the only argument to `f`.
- The pre-compiler may assume that some operators are associative, for optimizations. This may cause problems if these operators have non-associative tag methods.
- Old pre-compiled code is obsolete, and must be re-compiled.

Changes in the Libraries

- When traversing a table with `next` or `foreach`, the table cannot be modified in any way.
- General read patterns are now obsolete.
- The functions `rawgettable` and `rawsettable` have been renamed to `rawget` and `rawset`.
- The functions `foreachvar`, `nextvar`, `rawsetglobal`, and `rawgetglobal` are obsolete. You can get their functionality using table operations over the table of globals, which is returned by `globals`.
- `setglobal` and `sort` no longer return a value; `type` no longer returns a second value.
- The `p` option in function `call` is now obsolete.

Changes in the API

- The API has been completely rewritten: It is now fully reentrant and much clearer.
- The debug API has been completely rewritten.

The Complete Syntax of Lua

chunk	→	{*stat* [‘;’]}
block	→	*chunk*
stat	→	*varlist1* ‘=’ *explist1*
	|	*functioncall*
	|	**do** *block* **end**
	|	**while** *exp1* **do** *block* **end**
	|	**repeat** *block* **until** *exp1*
	|	**if** *exp1* **then** *block* {**elseif** *exp1* **then** *block*} [**else** *block*] **end**
	|	**return** [*explist1*]
	|	**break**
	|	**for** *name* ‘=’ *exp1* ‘,’ *exp1* [‘,’ *exp1*] **do** *block* **end**
	|	**for** *name* ‘,’ *name* **in** *exp1* **do** *block* **end**
	|	**function** *funcname* ‘(’ [*parlist1*] ‘)’ *block* **end**
	|	**local** *declist* [*init*]
funcname	→	*name* | *name* ‘.’ *name* | *name* ‘:’ *name*
varlist1	→	*var* {‘,’ *var*}
var	→	*name* | *varorfunc* ‘[’ *exp1* ‘]’ | *varorfunc* ‘.’ *name*
varorfunc	→	*var* | *functioncall*
declist	→	*name* {‘,’ *name*}
init	→	‘=’ *explist1*
explist1	→	{*exp1* ‘,’} *exp*
exp1	→	*exp*
exp	→	**nil** | *number* | *literal* | *var* | *function* | *upvalue*
	|	*functioncall* | *tableconstructor* | ‘(’ *exp* ‘)’ | *exp binop exp* | *unop exp*
functioncall	→	*varorfunc args* | *varorfunc* ‘:’ *name args*
args	→	‘(’ [*explist1*] ‘)’ | *tableconstructor* | *literal*
function	→	**function** ‘(’ [*parlist1*] ‘)’ *block* **end**
parlist1	→	‘...’ | *name* {‘,’ *name*} [‘,’ ‘...’]
upvalue	→	‘%’ *name*
tableconstructor	→	‘{’ *fieldlist* ‘}’
fieldlist	→	*lfieldlist* | *ffieldlist* | *lfieldlist* ‘;’ *ffieldlist* | *ffieldlist* ‘;’ *lfieldlist*
lfieldlist	→	[*lfieldlist1*]
ffieldlist	→	[*ffieldlist1*]
lfieldlist1	→	*exp* {‘,’ *exp*} [‘,’]
ffieldlist1	→	*ffield* {‘,’ *ffield*} [‘,’]
ffield	→	‘[’ *exp* ‘]’ ‘=’ *exp* | *name* ‘=’ *exp*
binop	→	‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘..’
	|	‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’
	|	**and** | **or**
unop	→	‘-’ | **not**

Index