Reference Manual of the Programming Language Lua 3.2

Roberto Ierusalimschy Luiz Henrique de Figueiredo Waldemar Celes

lua@tecgraf.puc-rio.br

TeCGraf — Computer Science Department — PUC-Rio

$Date: 1999/05/27 20:21:03

Abstract

Lua is a programming language originally designed for extending applications, but 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 3.2 of the Lua programming language and the API that allows interaction between Lua programs and their host C programs.

Sumário

Lua é uma linguagem de programação originalmente projetada para extensão de aplicações, e que é também frequentemente usada como uma linguagem de propósito geral. Lua combina uma sintaxe procedural simples (similar a Pascal) com poderosas facilidades para descrição de dados baseadas em tabelas associativas e uma semântica estensível. Lua tem tipagem dinâmica, é interpretada via bytecodes, e tem gerenciamento automático de memória com coleta de lixo, tornando-se ideal para configuração, scripting, e prototipagem rápida.

Este documento descreve a versão 3.2 da linguagem de programação Lua e a Interface de Programação (API) que permite a interação entre programas Lua e programas C hospedeiros.

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.

This implementation contains no third-party code.

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 light-weight, but powerful, 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 provided as usual with no guarantees, as stated in the 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/lua.tar.gz
```

2 Environment and Chunks

All statements in Lua are executed in a *global environment*. This environment, which keeps all global variables, 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. Optionally, a user can create multiple independent global environments (see Section 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 do not need declaration. Any variable is assumed to be global unless explicitly declared local (see Section 4.5.5). Before the first assignment, the value of a global variable is **nil**; this default can be changed (see Section 4.8).

The unit of execution of Lua is called a *chunk*. A chunk is simply a sequence of statements:

$$\mathit{chunk} \quad \rightarrow \quad \{\mathit{stat}\}\ [\mathit{ret}]$$

Statements are described in Section 4.5. (As usual, $\{a\}$ means 0 or more a's, $[a]$ means an optional a and $\{a\}^+$ means one or more a's.)

A chunk may be in a file or in a string inside the host program. A chunk may optionally end with a `return` statement (see Section 4.5.3). When a chunk is executed, first all its code is pre-compiled, then the statements are executed in sequential order. All modifications a chunk effects on the global environment persist after the chunk end.

Chunks may also be pre-compiled into binary form; 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. 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 eight-bit clean, and so strings may contain any 8-bit character, *including* embedded zeros (`'\0'`). The function `type` returns a string describing the type of a given value (see Section 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. They 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 type *userdata* is provided to allow arbitrary C pointers to be stored in Lua variables. It corresponds to a `void*` and has no pre-defined operations in Lua, besides assignment and equality test. However, by using *tag methods*, the programmer can define operations for *userdata* values (see Section 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, 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 itself as the first parameter (see Section 4.6.9).

Note that tables are *objects*, and not values. Variables cannot 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 Section 4.6.7).

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 Section 4.8). Each of the types *nil*, *number* and *string* has a different tag. All values of each of these types have this same pre-defined tag. 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 as many different tags as needed (see Section 4.8). Tags are created with the function `newtag`, and the function `tag` returns the tag of a given value. To change the tag of a given table, there is the function `settag` (see Section 6.1).

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. 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       do        else      elseif
end       function  if        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 other different identifiers. As a convention, identifiers starting with underscore followed by uppercase letters 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'` (new line), `'\r'` (carriage return), `'\t'` (horizontal tab), `'\v'` (vertical tab), `'\\'`, (backslash), `'\"'`, (double quote), and `'\''` (single quote). 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 0.

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 Section 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 The Pre-processor

All lines that start with a `$` sign are handled by a pre-processor. The `$` sign must be immediately followed by one of the following directives:

`debug` — turn on debugging facilities (see Section 4.9).

`nodebug` — turn off debugging facilities (see Section 4.9).

`if` ***cond*** — starts a conditional part. If *cond* is false, then this part is skipped by the lexical analyzer.

`ifnot` ***cond*** — starts a conditional part. If *cond* is true, then this part is skipped by the lexical analyzer.

`end` — ends a conditional part.

`else` — starts an "else" conditional part, flipping the "skip" status.

`endinput` — ends the lexical parse of the file.

Directives may be freely nested. Particularly, a `$endinput` may occur inside a `$if`; in that case, even the matching `$end` is not parsed.

A *cond* part may be

`nil` — always false.

`1` — always true.

name — true if the value of the global variable *name* is different from **nil**. Note that *name* is evaluated *before* the chunk starts its execution. Therefore, actions in a chunk do not affect its own conditional directives.

4.3 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. For complete control on how numbers are converted to strings, use the `format` function (see Section 6.2).

4.4 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 a 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 more needs than values, then the list is extended with as many **nil**'s as needed. Adjustment occurs in multiple assignment (see Section 4.5.2) and function calls (see Section 4.6.8).

4.5 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 Section 4.6.7), and local variable declarations (see Section 4.5.5).

4.5.1 Blocks

A block is a list of statements, which are executed sequentially. A statement may be optionally followed by a semicolon:

$$\begin{array}{rcl} \mathit{block} & \rightarrow & \{\mathit{stat}\ \mathit{sc}\}\ [\mathit{ret}] \\ \mathit{sc} & \rightarrow & [\text{';'}] \end{array}$$

For syntactic reasons, a `return` statement can only be written as the last statement of a block. This restriction also avoids some "statement not reached" conditions.

A block may be explicitly delimited:

$$\mathit{stat} \ \rightarrow \ \textbf{do}\ \mathit{block}\ \textbf{end}$$

This is useful to control the scope of local variables (see Section 4.5.5).

4.5.2 Assignment

The language 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. Both lists have their elements separated by commas:

$$\begin{array}{rcl} \textit{stat} & \rightarrow & \textit{varlist1} \text{ '=' } \textit{explist1} \\ \textit{varlist1} & \rightarrow & \textit{var} \ \{\text{','} \ \textit{var}\} \end{array}$$

This statement first evaluates all values on the right side and eventual indices on the left side, and then makes the assignments. Therefore, it can be used to exchange two values, as in

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

The two lists may have different lengths. Before the assignment, the list of values is *adjusted* to the length of the list of variables (see Section 4.4).

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

$$\textit{var} \ \rightarrow \ \textit{name}$$

Square brackets are used to index a table:

$$\textit{var} \ \rightarrow \ \textit{simpleexp} \text{ '[' } \textit{exp1} \text{ ']'}$$

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

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

$$\textit{var} \ \rightarrow \ \textit{simpleexp} \text{ '.' } \textit{name}$$

The meaning of assignments and evaluations of global variables and indexed variables can be changed by tag methods (see Section 4.8). Actually, an assignment `x = val`, where `x` is a global variable, is equivalent to a call `setglobal('x', val)`; an assignment `t[i] = val` is equivalent to `settable_event(t, i, val)`. See Section 4.8 for a complete description of these functions. (Function `setglobal` is pre-defined in Lua. Function `settable_event` is used only for explanatory purposes.)

4.5.3 Control Structures

The condition expression of a control structure may return any value. All values different from **nil** are considered true; only **nil** is considered false. `if`'s, `while`'s and `repeat`'s have the usual meaning.

$$\begin{array}{rcl} \textit{stat} & \rightarrow & \textbf{while} \ \textit{exp1} \ \textbf{do} \ \textit{block} \ \textbf{end} \\ & | & \textbf{repeat} \ \textit{block} \ \textbf{until} \ \textit{exp1} \\ & | & \textbf{if} \ \textit{exp1} \ \textbf{then} \ \textit{block} \ \{\textbf{elseif} \ \textit{exp1} \ \textbf{then} \ \textit{block}\} \ [\textbf{else} \ \textit{block}] \ \textbf{end} \end{array}$$

A `return` is used to return values from a function or from a chunk. Because they may return more than one value, the syntax for a return statement is

$$\textit{ret} \ \rightarrow \ \textbf{return} \ [\textit{explist1}] \ [\textit{sc}]$$

4.5.4 Function Calls as Statements

Because of possible side-effects, function calls can be executed as statements:

$$stat \rightarrow functioncall$$

In this case, all returned values are thrown away. Function calls are explained in Section 4.6.8.

4.5.5 Local Declarations

Local variables may be declared anywhere inside a block. Their scope begins after the declaration and lasts until the end of the block. The declaration may include an initial assignment:

$$\begin{aligned} stat &\rightarrow \textbf{local}\ declist\ [init] \\ declist &\rightarrow name\ \{\text{','}\ name\} \\ init &\rightarrow \text{'='}\ explist1 \end{aligned}$$

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

4.6 Expressions

4.6.1 Basic Expressions

Basic expressions are

$$\begin{aligned} exp &\rightarrow \text{'('}\ exp\ \text{')'} \\ exp &\rightarrow \textbf{nil} \\ exp &\rightarrow \text{'number'} \\ exp &\rightarrow \text{'literal'} \\ exp &\rightarrow function \\ exp &\rightarrow simpleexp \end{aligned}$$

$$\begin{aligned} simpleexp &\rightarrow var \\ simpleexp &\rightarrow upvalue \\ simpleexp &\rightarrow functioncall \end{aligned}$$

Numbers (numerical constants) and string literals are explained in Section 4.1; variables are explained in Section 4.5.2; upvalues are explained in Section 4.7; function definitions (*function*) are explained in Section 4.6.9; function calls are explained in Section 4.6.8.

An access to a global variable `x` is equivalent to a call `getglobal('x')`; an access to an indexed variable `t[i]` is equivalent to a call `gettable_event(t, i)`. See Section 4.8 for a description of these functions. (Function `getglobal` is pre-defined in Lua. Function `gettable_event` is used only for explanatory purposes.)

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

$$exp1 \rightarrow exp$$

4.6.2 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 Section 4.3), then all operations except exponentiation have the usual meaning. Otherwise, an appropriate tag method is called (see Section 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 Section 6.3).

4.6.3 Relational Operators

Lua provides the following relational operators:

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

All these 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 that the conversion rules of Section 4.3 *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 other 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 "order" tag method is called (see Section 4.8).

4.6.4 Logical Operators

The logical operators are

```
and   or   not
```

Like control structures, all logical operators consider **nil** as false and anything else as true. The operator `and` returns **nil** if its first argument is **nil**; otherwise, it returns its second argument. The 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 when necessary.

A useful Lua idiom is `x = x or v`, which is equivalent to

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

i.e., it sets `x` to a default value `v` when `x` is not set.

4.6.5 Concatenation

The string concatenation operator in Lua is denoted by "`..`". If both operands are strings or numbers, they are converted to strings according to the rules in Section 4.3. Otherwise, the "concat" tag method is called (see Section 4.8).

4.6.6 Precedence

Operator precedence 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.

4.6.7 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 fields.

The general syntax for constructors is

$$
\begin{array}{rcl}
\textit{tableconstructor} & \rightarrow & \text{'\{'}\ \textit{fieldlist}\ \text{'\}'} \\
\textit{fieldlist} & \rightarrow & \textit{lfieldlist}\ \mid \textit{ffieldlist}\ \mid \textit{lfieldlist}\ \text{';'}\ \textit{ffieldlist}\ \mid \textit{ffieldlist}\ \text{';'}\ \textit{lfieldlist} \\
\textit{lfieldlist} & \rightarrow & [\textit{lfieldlist1}] \\
\textit{ffieldlist} & \rightarrow & [\textit{ffieldlist1}]
\end{array}
$$

The form *lfieldlist1* is used to initialize lists:

$$\textit{lfieldlist1} \rightarrow \textit{exp}\ \{\text{','}\ \textit{exp}\}\ [\text{','}]$$

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:

$$
\begin{array}{rcl}
\textit{ffieldlist1} & \rightarrow & \textit{ffield}\ \{\text{','}\ \textit{ffield}\}\ [\text{','}] \\
\textit{ffield} & \rightarrow & \text{'['}\ \textit{exp}\ \text{']'}\ \text{'='}\ \textit{exp}\ \mid \textit{name}\ \text{'='}\ \textit{exp}
\end{array}
$$

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-collon. 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,}
```

4.6.8 Function Calls

A function call has the following syntax:

functioncall → *simpleexp args*

First, *simpleexp* 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 *simpleexp*, and then the original call parameters.

The form:

functioncall → *simpleexp* ':' *name args*

can be used to call "methods". A call `simpleexp:name(...)` is syntactic sugar for

```
simpleexp.name(simpleexp, ...)
```

except that `simpleexp` is evaluated only once.

Arguments have the following syntax:

args → '(' [*explist1*] ')'
args → *tableconstructor*
args → 'literal'
explist1 → *exp1* {',' *exp1*}

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

Because a function can return any number of results (see Section 4.5.3), the number of results must be adjusted before used. If the function is called as a statement (see Section 4.5.4), 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. Note that the only place that can hold many values is the last (or the only) expression in an assignment or in a return statement; see examples below.

```
f();                -- adjusted to 0
g(x, f());          -- f() is adjusted to 1
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()
```

4.6.9 Function Definitions

The syntax for function definition is

function → **function** '(' [*parlist1*] ')' *block* **end**
stat → **function** *funcname* '(' [*parlist1*] ')' *block* **end**
funcname → *name* | *name* '.' *name*

The statement

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

is just syntactic sugar for

```
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 Section 4.7), and the function is *instantiated* (or "closed"). This function instance (or "closure") is the final value of the expression. Different instances of a 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 Section 4.4), unless the function is a *vararg* function, indicated by the dots (...) at the end of its parameter list. A vararg function does not adjust its argument list; instead, it collects any extra arguments into an implicit parameter, called `arg`. This parameter is always initialized as a table, with a field `n` whose value is the number of extra arguments, and the extra arguments at positions 1, 2, ...

As an example, suppose definitions like:

```
function f(a, b) end
function g(a, b, ...) 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

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}
```

Results are returned using the **`return`** statement (see Section 4.5.3). If control reaches the end of a function without a return instruction, then the function returns with no results.

There is a special syntax for defining methods, that is, functions that have an implicit extra parameter `self`:

function → **function** *name* ':' *name* '(' [*parlist1*] ')' *block* **end**

Thus, a declaration like

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

is equivalent to

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

that is, the function gets an extra formal parameter called `self`. Note that the variable `v` must have been previously initialized with a table value.

4.7 Visibility and Upvalues

A function body may refer to its own local variables (which includes its parameters) and to global variables, as long as they are not shadowed by local variables 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*.

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.

Here are some examples:

```
a,b,c = 1,2,3   -- global variables
function f (x)
  local b       -- x and b are local to f
  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')
    p = %c      -- OK, access frozen value of global 'c'
    p = %y      -- ERROR: 'y' is not visible where 'g' is defined
  end           -- g
end             -- f
```

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 evaluation of a 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 Section 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 function returns the previous tag method for that pair. Another 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. The function not only shows when a tag method is called, but also its arguments, its results and the default behavior. Please notice that 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 (`rawgetglobal`, `tonumber`, `call`, etc.) are described in Section 6.1.

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

The function `getbinmethod` 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
```

```
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 is applied.

```
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 an 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 a `<` operation is applied to non numerical or non string operands.

```
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
```

"gt": called when a > operation is applied to non numerical or non string operands. Behavior similar to the "lt" event.

"le": called when a <= operation is applied to non numerical or non string operands. Behavior similar to the "lt" event.

"ge": called when a >= operation is applied to non numerical or non string operands. Behavior similar to the "lt" event.

"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 event "gettable" 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`.

```
function getglobal (varname)
  local value = rawgetglobal(varname)
  local tm = gettagmethod(tag(value), "getglobal")
  if not tm then
    return value
  else
    return tm(varname, value)
```

```
    end
  end
```

The function `getglobal` is pre-defined in Lua (see Section 6.1).

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

```
  function setglobal (varname, newvalue)
    local oldvalue = rawgetglobal(varname)
    local tm = gettagmethod(tag(oldvalue), "setglobal")
    if not tm then
      return rawsetglobal(varname, newvalue)
    else
      return tm(varname, oldvalue, newvalue)
    end
  end
```

Notice: the function `setglobal` is pre-defined in Lua (see Section 6.1).

"gettable": called whenever Lua accesses an indexed variable. This method cannot be set for tables with 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 = rawgettable(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 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
      rawsettable(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
      local i = arg.n
      while i > 0 do
        arg[i+1] = arg[i]
        i = i-1
      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" an object. This method cannot be set for strings, numbers, functions, and userdata with default tag. For each object 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
```

Moreover, at the end of a garbage collection cycle, Lua does the equivalent of the call `gc_event(nil)`.

4.9 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, 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_callfunction`) is terminated, returning an error condition.

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 Section 6.1). The standard I/O library redefines `_ERRORMESSAGE`, and uses the debug facilities (see Section 7) to print some extra information, such as the call stack.

To provide more information about errors, Lua programs should include the compilation pragma `$debug`. When an error occurs in a chunk compiled with this option, the I/O error routine is able to print the number of the lines where the calls (and the error) were made.

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

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 the Lua library. The API functions can be classified in the following categories:

1. managing states;
2. exchanging values between C and Lua;
3. executing Lua code;
4. manipulating (reading and writing) Lua objects;
5. calling Lua functions;
6. C functions to be called by Lua;
7. manipulating references to Lua Objects.

All API functions and related types and constants are declared in the header file `lua.h`.

5.1 Managing States

The whole state of the Lua interpreter (global variables, stack, tag methods, etc) is stored in a dynamic structure pointed by

```
typedef struct lua_State lua_State;
extern lua_State *lua_state;
```

The variable `lua_state` is the only C global variable in the Lua library.

Before calling any API function, this state must be initialized. This is done by calling

```
void lua_open (void);
```

This function allocates and initializes some internal structures, and defines all pre-defined functions of Lua. If `lua_state` is already different from `NULL`, `lua_open` has no effect; therefore, it is safe to call this function multiple times. All standard libraries call `lua_open` when they are opened.

Function `lua_setstate` is used to change the current state of Lua:

```
lua_State *lua_setstate (lua_State *st);
```

It sets `lua_state` to `st` and returns the old state.

Multiple, independent states may be created. For that, you must set `lua_state` back to `NULL` before calling `lua_open`. An easy way to do that is defining an auxiliary function:

```
lua_State *lua_newstate (void) {
  lua_State *old = lua_setstate(NULL);
  lua_open();
  return lua_setstate(old);
}
```

This function creates a new state without changing the current state of the interpreter. Note that any new state is created with all predefined functions, but any additional library (such as the standard libraries) must be explicitly open in the new state, if needed.

If necessary, a state may be released by calling

```
void lua_close (void);
```

This function destroys all objects in the current Lua environment (calling the correspondent garbage collector tag methods), frees all dynamic memory used by the state, and then sets `lua_state` to `NULL`. Usually, there is no need to call this function, since these resources are naturally released when the program ends. If `lua_state` is already `NULL`, `lua_close` has no effect.

If you are using multiple states, you may find useful to define the following function, which releases a given state:

```
void lua_freestate (lua_State *st) {
  lua_State *old = lua_setstate(st);
  lua_close();
  if (old != st) lua_setstate(old);
}
```

5.2 Exchanging Values between C and Lua

Because Lua has no static type system, all values passed between Lua and C have type `lua_Object`, which works like an abstract type in C that can hold any Lua value. Values of type `lua_Object` have no meaning outside Lua; for instance, the comparison of two `lua_Object's` is undefined.

To check the type of a `lua_Object`, the following functions are available:

```
int lua_isnil       (lua_Object object);
int lua_isnumber    (lua_Object object);
int lua_isstring    (lua_Object object);
int lua_istable     (lua_Object object);
int lua_isfunction  (lua_Object object);
int lua_iscfunction (lua_Object object);
int lua_isuserdata  (lua_Object object);
```

These functions return 1 if the object is compatible with the given type, and 0 otherwise. The function `lua_isnumber` accepts numbers and numerical strings, whereas `lua_isstring` accepts strings and numbers (see Section 4.3), and `lua_isfunction` accepts Lua functions and C functions.

To get the tag of a `lua_Object`, the following function is available:

```
int lua_tag (lua_Object object);
```

To translate a value from type `lua_Object` to a specific C type, the programmer can use:

```
double         lua_getnumber    (lua_Object object);
char          *lua_getstring    (lua_Object object);
long           lua_strlen       (lua_Object object);
lua_CFunction  lua_getcfunction (lua_Object object);
void          *lua_getuserdata  (lua_Object object);
```

`lua_getnumber` converts a `lua_Object` to a floating-point number. This `lua_Object` must be a number or a string convertible to number (see Section 4.3); otherwise, `lua_getnumber` returns 0.

`lua_getstring` converts a `lua_Object` to a string (`char*`). This `lua_Object` must be a string or a number; otherwise, the function returns 0 (the `NULL` pointer). This function does not create a new string, but returns a pointer to a string inside the Lua environment. Those strings always have a 0 after their last character (like in C), but may contain other zeros in their body. If you do not know whether a string may contain zeros, you can use `lua_strlen` to get the actual length. Because Lua has garbage collection, there is no guarantee that the pointer returned by `lua_getstring` will be valid after the block ends (see Section 5.3).

`lua_getcfunction` converts a `lua_Object` to a C function. This `lua_Object` must have type *CFunction*; otherwise, `lua_getcfunction` returns 0 (the `NULL` pointer). The type `lua_CFunction` is explained in Section 5.7.

`lua_getuserdata` converts a `lua_Object` to `void*`. This `lua_Object` must have type *userdata*; otherwise, `lua_getuserdata` returns 0 (the `NULL` pointer).

5.3 Garbage Collection

Because Lua has automatic memory management and garbage collection, a `lua_Object` has a limited scope, and is only valid inside the *block* where it has been created. A C function called from Lua is a block, and its parameters are valid only until its end. It is good programming practice to convert Lua objects to C values as soon as they are available, and never to store `lua_Objects` in C global variables.

A garbage collection cycle can be forced by:

```
long lua_collectgarbage (long limit);
```

This function returns the number of objects collected. The argument `limit` makes the next cycle occur only after that number of new objects have been created. If `limit`=0, then Lua uses an adaptive heuristics to set this limit.

All communication between Lua and C is done through two abstract data types, called *lua2C* and *C2lua*. The first one, as the name implies, is used to pass values from Lua to C: parameters when Lua calls C and results when C calls Lua. The structure C2lua is used in the reverse direction: parameters when C calls Lua and results when Lua calls C.

The structure lua2C is an abstract array, which can be indexed with the function:

```
lua_Object lua_lua2C (int number);
```

where `number` starts with 1. When called with a number larger than the array size, this function returns `LUA_NOOBJECT`. In this way, it is possible to write C functions that receive a variable number of parameters, and to call Lua functions that return a variable number of results. Note that the structure lua2C cannot be directly modified by C code.

The second structure, C2lua, is an abstract stack. Pushing elements into this stack is done with the following functions:

```
void lua_pushnumber    (double n);
void lua_pushlstring   (char *s, long len);
void lua_pushstring    (char *s);
void lua_pushusertag   (void *u, int tag);
void lua_pushnil       (void);
void lua_pushobject    (lua_Object object);
void lua_pushcfunction (lua_CFunction f);  /* macro */
```

All of them receive a C value, convert it to a corresponding `lua_Object`, and leave the result on the top of C2lua. In particular, functions `lua_pushlstring` and `lua_pushstring` make an internal copy of the given string. Function `lua_pushstring` can only be used to push proper C strings (that is, strings that do not contain zeros and end with a zero); otherwise you should use the more generic `lua_pushlstring`. The function

```
lua_Object lua_pop (void);
```

returns a reference to the object at the top of the C2lua stack, and pops it.

As a general rule, all API functions pop from the stack all elements they use.

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, 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 (void);
```

The function `lua_settag` changes the tag of the object on the top of C2lua (and pops it); the object must be a userdata or a table.

```
void lua_settag (int tag);
```

`tag` must be a value created with `lua_newtag`.

When C code calls Lua repeatedly, as in a loop, objects returned by these calls can accumulate, and may cause a stack overflow. To avoid this, nested blocks can be defined with the functions:

```
void lua_beginblock (void);
void lua_endblock   (void);
```

After the end of the block, all `lua_Object`'s created inside it are released. The use of explicit nested blocks is good programming practice and is strongly encouraged.

5.4 Executing Lua Code

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

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

All these functions return an error code: 0, in case of success; non zero, in case of errors. More specifically, `lua_dofile` returns 2 if for any reason it could not open the file. When called with argument `NULL`, `lua_dofile` executes the `stdin` stream. Functions `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`). Function `lua_dostring` executes only source code.

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

These functions return, in structure lua2C, any values eventually returned by the chunks. They also empty the stack C2lua.

5.5 Manipulating Lua Objects

To read the value of any global Lua variable, one uses the function:

```
lua_Object lua_getglobal (char *varname);
```

As in Lua, this function may trigger a tag method. To read the real value of any global variable, without invoking any tag method, use the *raw* version:

```
lua_Object lua_rawgetglobal (char *varname);
```

To store a value previously pushed onto C2lua in a global variable, there is the function:

```
void lua_setglobal (char *varname);
```

As in Lua, this function may trigger a tag method. To set the real value of any global variable, without invoking any tag method, use the *raw* version:

```
void lua_rawsetglobal (char *varname);
```

Tables can also be manipulated via the API. The function

```
lua_Object lua_gettable (void);
```

pops a table and an index from the stack C2lua, and returns the contents of the table at that index. As in Lua, this operation may trigger a tag method. To get the real value of any table index, without invoking any tag method, use the *raw* version:

```
lua_Object lua_rawgettable (void);
```

To store a value in an index, the program must push the table, the index, and the value onto C2lua, and then call the function

```
void lua_settable (void);
```

Again, the tag method for "settable" may be called. To set the real value of any table index, without invoking any tag method, use the *raw* version:

```
void lua_rawsettable (void);
```

Finally, the function

```
lua_Object lua_createtable (void);
```

creates and returns a new, empty table.

5.6 Calling Lua Functions

Functions defined in Lua by a chunk can be called from the host program. This is done using the following protocol: first, the arguments to the function are pushed onto C2lua (see Section 5.3), in direct order, i.e., the first argument is pushed first. Then, the function is called using

```
int lua_callfunction (lua_Object function);
```

This function returns an error code: 0, in case of success; non zero, in case of errors. Finally, the results (a Lua function may return many values) are returned in structure lua2C, and can be retrieved with the macro `lua_getresult`, which is just another name to function `lua_lua2C`. Note that function `lua_callfunction` pops all elements from the C2lua stack.

The following example shows how a C program may do the equivalent to the Lua code:

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

```
lua_pushstring("how");                               /* 1st argument */
lua_pushobject(lua_getglobal("t"));      /* push value of global 't' */
lua_pushstring("x");                          /* push the string 'x' */
lua_pushobject(lua_gettable());      /* push result of t.x (2nd arg) */
lua_pushnumber(4);                                   /* 3rd argument */
lua_callfunction(lua_getglobal("f"));           /* call Lua function */
lua_pushobject(lua_getresult(1));   /* push first result of the call */
lua_setglobal("a");                       /* set global variable 'a' */
lua_pushobject(lua_getresult(2));  /* push second result of the call */
lua_setglobal("b");                       /* set global variable 'b' */
```

Some special Lua functions have exclusive interfaces. A C function can generate a Lua error calling the function

```
void lua_error (char *message);
```

This function never returns. If the C function 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(1)`. The `message` is passed to the error handler function, `_ERRORMESSAGE`. If `message` is `NULL`, then `_ERRORMESSAGE` is not called.

Tag methods can be changed with:

```
lua_Object lua_settagmethod (int tag, char *event);
```

The first parameter is the tag, and the second is the event name (see Section 4.8); the new method is pushed from C2lua. This function returns a `lua_Object`, which is the old tag method value. To get just the current value of a tag method, use the function

```
lua_Object lua_gettagmethod (int tag, char *event);
```

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

```
int lua_copytagmethods (int tagto, int tagfrom);
```

This function returns `tagto`.

5.7 C Functions

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

```
#define lua_register(n,f)       (lua_pushcfunction(f), lua_setglobal(n))
/* 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 void (*lua_CFunction) (void);
```

that is, a pointer to a function with no parameters and no results.

In order to communicate properly with Lua, a C function must follow a protocol, which defines the way parameters and results are passed.

A C function receives its arguments in structure lua2C; to access them, it uses the macro `lua_getparam`, again just another name for `lua_lua2C`. To return values, a C function just pushes them onto the stack C2lua, in direct order (see Section 5.2). Like a Lua function, a C function called by Lua can also return many results.

When a C function is created, it is possible to associate some *upvalues* to it, thus creating a C closure; then these values are passed to the function whenever it is called, as common arguments. To associate upvalues to a function, first these values must be pushed on C2lua. Then the function

```
void lua_pushcclosure (lua_CFunction fn, int n);
```

is used to put the C function on C2lua, with the argument `n` telling how many upvalues must be associated with the function; in fact, the macro `lua_pushcfunction` is defined as `lua_pushcclosure` with `n` set to 0. Then, any time the function is called, these upvalues are inserted as the first arguments to the function, before the actual arguments provided in the call.

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

5.8 References to Lua Objects

As noted in Section 5.3, `lua_Objects` are volatile. If the C code needs to keep a `lua_Object` outside block boundaries, then it must create a *reference* to the object. The routines to manipulate references are the following:

```
int        lua_ref    (int lock);
lua_Object lua_getref (int ref);
void       lua_unref  (int ref);
```

The function `lua_ref` creates a reference to the object that is on the top of the stack, and returns this reference. If `lock` is true, the object is *locked*: this means the object will not be garbage collected. Note that an unlocked reference may be garbage collected. Whenever the referenced object is needed, a call to `lua_getref` returns a handle to it; if the object has been collected, `lua_getref` returns `LUA_NOOBJECT`.

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

6 Predefined Functions and Libraries

The set of predefined functions in Lua is small but powerful. Most of them provide features that allow some degree of reflexivity in the language. Some of these features cannot be simulated with the rest of the language nor with the standard Lua API. Others are just convenient interfaces to common API functions.

The libraries, on the other hand, provide useful routines that are implemented directly through the standard API. Therefore, they are not necessary to the language, and are provided as separate C modules. Currently, there are three standard libraries:

- 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_strlibopen`, `lua_mathlibopen`, and `lua_iolibopen`, declared in `lualib.h`.

6.1 Predefined Functions

• `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 Section 6.1).

By default, all results from `func` are just returned by the call. If the string `mode` contains `"p"`, the results are *packed* in a single table. That is, `call` returns just one table; at index `n`, the table has the total number of results from the call; the first result is at index 1, etc. For instance, the following calls produce the following results:

```
a = call(sin, {5})                --> a = 0.0871557 = sin(5)
a = call(max, {1,4,5; n=2})       --> a = 4 (only 1 and 4 are arguments)
a = call(max, {1,4,5; n=2}, "p")  --> a = {4; n=1}
t = {x=1}
a = call(next, {t,nil;n=2}, "p")  --> a={"x", 1; n=2}
```

By default, if an error occurs during the function call, 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 provided, `errhandler` is temporarily set as the error function `_ERRORMESSAGE`, while `func` runs. In particular, if `errhandler` is **nil**, no error messages will be issued during the execution of the called function.

• `collectgarbage ([limit])`

Forces a garbage collection cycle. Returns the number of objects collected. An optional argument, `limit`, is a number that makes the next cycle occur only after that number of new objects have been created. If `limit` is absent or equal to 0, Lua uses an adaptive algorithm to set this limit. `collectgarbage` is equivalent to the API function `lua_collectgarbage`.

• `dofile (filename)`

Receives a file name, opens the file, and executes the file 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. `dofile` is equivalent to the API function `lua_dofile`.

• `dostring (string [, chunkname])`

Executes a given string as a Lua chunk. If there is any error executing the string, `dostring` returns **nil**. Otherwise, it returns the values returned by the chunk, or a non **nil** value if the chunk returns no values. An optional second parameter (`chunkname`) is the "name of the chunk", used in error messages and debug information. `dostring` is equivalent to the API function `lua_dostring`.

• `newtag ()`

Returns a new tag. `newtag` is equivalent to the API function `lua_newtag`.

• `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. It returns the next index of the table and the value associated with the index. When called with **nil** as its second argument, the function returns the first index of the table (and its associated value). When called with the last index, or with **nil** in an empty table, it returns **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, the function 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 counter or the function `foreachi`). If the table indices are modified in any way during a traversal, the semantics of `next` is undefined.

This function cannot be written with the standard API.

• `nextvar (name)`

This function is similar to the function `next`, but iterates instead over the global variables. Its single argument is the name of a global variable, or **nil** to get a first name. Similarly to `next`, it returns the name of another variable and its value, or **nil** if there are no more variables. There can be no creation of new global variables during the traversal; otherwise the semantics of `nextvar` is undefined.

This function cannot be written with the standard API.

• `tostring (e)`

Receives an argument of any type and converts it to a string in a reasonable format. For complete control on how numbers are converted, use function `format`.

• `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 Section 6.4 for functions for formatted output.

• `_ALERT (message)`

Prints its only string argument to `stderr`. All error messages in Lua are printed through this function. Therefore, a program may redefine it to change the way such messages are shown (for instance, for systems without `stderr`).

• `tonumber (e [, base])`

Receives one argument, and tries to convert it 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 Section 4.3). In other bases, only integers are accepted.

• `type (v)`

Allows Lua 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"`.

• `tag (v)`

Allows Lua to test the tag of a value (see Section 3). It receives one argument, and returns its tag (a number). `tag` is equivalent to the API function `lua_tag`.

• `settag (t, tag)`

Sets the tag of a given table (see Section 3). `tag` must be a value created with `newtag` (see Section 6.1). It returns the value of its first argument (the table). For security reasons, it is impossible to change the tag of a userdata from Lua.

• `assert (v [, message])`

Issues an *"assertion failed!"* error when its argument 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
```

• `error (message)`

Calls the error handler 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**, the error handler is not called. Function `error` never returns. `error` is equivalent to the API function `lua_error`.

• `rawgettable (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**.

• `rawsettable (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.

• `rawsetglobal (name, value)`

Assigns the given value to a global variable. The string `name` does not need to be a syntactically valid variable name. Therefore, this function can set global variables with strange names like `"m v 1"` or `34`. Function `rawsetglobal` returns the value of its second argument.

• `setglobal (name, value)`

Assigns the given value to a global variable, or calls a tag method. Its full semantics is explained in Section 4.8. The string `name` does not need to be a syntactically valid variable name. Function `setglobal` returns the value of its second argument.

• `rawgetglobal (name)`

Retrieves the value of a global variable. The string `name` does not need to be a syntactically valid variable name.

• `getglobal (name)`

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

• `settagmethod (tag, event, newmethod)`

Sets a new tag method to the given pair *(tag, event)*. It returns the old method. If `newmethod` is **nil**, `settagmethod` restores the default behavior for the given event.

• `gettagmethod (tag, event)`

Returns the current tag method for a given pair *(tag, event)*.

• `copytagmethods (tagto, tagfrom)`

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

• `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 is its "size". 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
  local i = next(t, nil)
  while i do
    if type(i) == 'number' and i>max then max=i end
    i = next(t, i)
  end
  return max
end
```

• `foreach (table, function)`

Executes the given `function` 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, the loop is broken, and the value is returned as the final value of `foreach`.

This function could be defined in Lua:

```
function foreach (t, f)
  local i, v = next(t, nil)
  while i do
    local res = f(i, v)
    if res then return res end
    i, v = next(t, i)
  end
end
```

• `foreachi (table, function)`

Executes the given `function` 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 Section 6.1). If the function returns any non-**nil** value, the loop is broken, and the value is returned as the final value of `foreachi`.

This function could be defined in Lua:

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

• `foreachvar (function)`

Executes `function` over all global variables. For each variable, the function is called with its name and its value as arguments. If the function returns any non-nil value, the loop is broken, and the value is returned as the final value of `foreachvar`.

This function could be defined in Lua:

```
function foreachvar (f)
  local n, v = nextvar(nil)
  while n do
    local res = f(n, v)
    if res then return res end
    n, v = nextvar(n)
  end
end
```

• `tinsert (table [, pos] , value)`

Inserts element `value` at table position `pos`, shifting other elements to open space. The default value for `pos` is `n+1` (where `n` is the result of `getn(table)` (see Section 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 = n+1; value = arg[1]
  else
    pos = arg[1]; value = arg[2]
  end
  t.n = n+1;
  while n >= pos do
    t[n+1] = t[n]
    n = n-1
  end
  t[pos] = value
end
```

• `tremove (table [, pos])`

Removes from `table` the element at position `pos`, shifting other elements to close the space. Returns the value of the removed element. The default value for `pos` is `n` (where `n` is the result of `getn(table)` (see Section 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)
  pos = pos or n
  local value = t[pos]
  if n<=0 then return end
  while pos < n do
    t[pos] = t[pos+1]
    pos = pos+1
  end
  t[n] = nil
  t.n = n-1
  return value
end
```

• `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 Section 6.1). If `comp` is given, it must be a function that receives two table elements, and returns true when the first is less than the second (so that `not comp(a[i+1], a[i])` will be true after the sort). If `comp` is not given, the standard `<` Lua operator is used instead.

Function `sort` returns the (sorted) table.

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, the first character is at position 1 (not at 0, as in C).

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

Looks for the first *match* of `pattern` in `str`. If it finds one, then it returns the indices on `str` where this occurrence starts and ends; otherwise, it returns **nil**. If the pattern specifies captures, the captured strings are returned as extra results. A third optional numerical argument specifies where to start the search; its default value is 1. If `init` is negative, it is replaced by the length of the string minus its absolute value plus 1. Therefore, -1 points to the last character of `str`. A value of 1 as a fourth optional argument turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in `pattern` being considered "magic".

• `strlen (s)`

Receives a string and returns its length.

• `strsub (s, i [, j])`

Returns another string, which is a substring of `s`, starting at `i` and running until `j`. If `i` or `j` are negative, they are replaced by the length of the string minus their absolute value plus 1. Therefore, -1 points to the last character of `s` and -2 to the previous one. If `j` is absent, 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`.

• `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.

• `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.

• `strrep (s, n)`

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

• `strbyte (s [, i])`

Returns the internal numerical code of the character `s[i]`. If `i` is absent, then it is assumed to be 1. If `i` is negative, it is replaced by the length of the string minus its absolute value plus 1. Therefore, -1 points to the last character of `s`.

Note that 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 that numerical codes are not necessarily portable across platforms.

• `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`. This 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: function `format` *can only be used with strings that do not contain zeros.*

• **`gsub (s, pat, repl [, n])`**

Returns a copy of `s`, where all occurrences of the pattern `pat` have been replaced by a replacement string specified by `repl`. This function 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.

A 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="3.2"}
x = gsub("$name - $version", "%$(%w+)", function (v) return %t[v] end)
--> x="lua - 3.2"

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 character not in the list `^$()%.[]*+-?`) — 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 hexa-decimal 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 `()%.[]*-?`. It is strongly recommended that any control character (even the non magic), when used to represent itself in a pattern, should be preceded by a `%`.

`[char-set]` — Represents the class which is the union of all characters in char-set. To include a `]` in char-set, it must be the first character. A range of characters may be specified by separating the end characters of the range with a `-`. If `-` appears as the first or last character of char-set, then it represents itself. All classes `%`*x* described above can also be used as components in a char-set. All other characters in char-set represent themselves. E.g., assuming an *ascii* character set, `[%dA-Fa-f]` specifies the hexa-decimal digits.

`[^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 correspondent 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 more portable programs.

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*. That means that, if one reads the string from left to write, counting plus 1 for an x and minus 1 for a y, the ending y is the first 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.

Captures: a pattern may contain sub-patterns enclosed in parentheses, that 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.

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   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. 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. 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. Whenever a file handle is collected by the garbage collector, its correspondent stream is automatically closed.

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 string mode 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 string mode may also have a b at the end, which is needed in some systems to open the file in binary mode.

• `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.

> *System dependent*: 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.

> *System dependent*: if `filename` starts with a `|`, then a piped output 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 content of the file. 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, 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 removed when no longer needed.

• `read ([filehandle,] readpattern1, ...)`

Reads file `_INPUT`, or `filehandle` if this argument is given, according to read patterns, which specify how much to read. For each pattern, the function returns a string with the characters read, even if the pattern succeeds only partially, or **nil** if the read pattern fails *and* the result string would be empty. When called without patterns, it uses a default pattern that reads the next line (see below).

A *read pattern* is a sequence of read pattern items. An item may be a single character class or a character class followed by `?`, by `*`, or by `+`. A single character class reads the next character from the input if it belongs to the class, otherwise it fails. A character class followed by `?` reads the next character from the input if it belongs to the class; it never fails. A character class followed by `*` reads until a character that does not belong to the class, or end of file; since it can match a sequence of zero characters, it never fails. A character class followed by `+` reads until a character that does not belong to the class, or end of file; it fails if it cannot read at least one character. Note that the behavior of read patterns is slightly different from the regular pattern matching behavior, where a `*` expands to the maximum length *such that* the rest of the pattern does not fail. With the read pattern behavior there is no need for backtracking the reading.

A pattern item may contain sub-patterns enclosed in curly brackets, that describe *skips*. Characters matching a skip are read, but are not included in the resulting string.

There are some predefined patterns, as follows:

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

"*l" returns the next line (skipping the end of line), or **nil** on end of file. This is the default pattern. It is equivalent to the pattern `"[^\n]*{\n}"`.

"*a" reads the whole file. It is equivalent to the pattern `".*"`.

"*w" returns the next word (maximal sequence of non white-space characters), skipping spaces if necessary, or **nil** on end of file. It is equivalent to the pattern `"{%s*}%S+"`.

• `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.

• `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 locale.

• `clock ()`

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

• `exit ([code])`

Calls the C function `exit`, with an optional `code`, to terminate the program. The default value for `code` is 1.

• `getenv (varname)`

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

• `execute (command)`

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

• `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 Debugger 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 the header file `luadebug.h`.

7.1 Stack and Function Information

The main function to get information about the interpreter stack is

```
lua_Function lua_stackedfunction (int level);
```

It returns a handle (`lua_Function`) to the *activation record* of the function executing at a given level. Level 0 is the current running function, while level $n+1$ is the function that has called level n. When called with a level greater than the stack depth, `lua_stackedfunction` returns `LUA_NOOBJECT`.

The type `lua_Function` is just another name to `lua_Object`. Although, in this library, a `lua_Function` can be used wherever a `lua_Object` is required, when a parameter has type `lua_Function` it accepts only a handle returned by `lua_stackedfunction`.

Three other functions produce extra information about a function:

```
void lua_funcinfo (lua_Object func, char **source, int *linedefined);
int lua_currentline (lua_Function func);
char *lua_getobjname (lua_Object o, char **name);
```

`lua_funcinfo` gives the source and the line where the given function has been defined: 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. If the "function" is in fact the main code of a chunk, then `linedefined` is 0. If the function is a C function, then `linedefined` is -1, and `filename` is `"(C)"`.

The function `lua_currentline` gives the current line where a given function is executing. It only works if the function has been compiled with debug information. When no line information is available, `lua_currentline` returns -1.

The generation of debug information is controled by an internal flag, which can be switched with

```
int lua_setdebug (int debug);
```

This function sets the flag and returns its previous value. This flag can also be set from Lua (see Section 4.9).

Function `lua_getobjname` tries to find a reasonable name for a 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. Function `lua_getobjname` 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 `lua_getobjname` returns the string `"tag-method"`, and `name` is set to point to the event name. If the given function is the value of a global variable, then `lua_getobjname` returns the string `"global"`, and `name` points to the variable name. If the given function is neither a tag method nor a global variable, then `lua_getobjname` returns the empty string, and `name` is set to `NULL`.

7.2 Manipulating Local Variables

The following functions allow the manipulation of the local variables of a given activation record. They only work if the function has been compiled with debug information (see Section 4.9). Moreover, for these functions, a local variable becomes visible in the line after its definition.

```
lua_Object lua_getlocal (lua_Function func, int local_number, char **name);
int lua_setlocal (lua_Function func, int local_number);
```

`lua_getlocal` returns the value of a local variable, and sets `name` to point to the variable name. `local_number` is an index for local variables. The first parameter has index 1, and so on, until the last active local variable. When called with a `local_number` greater than the number of active local variables, or if the activation record has no debug information, `lua_getlocal` returns `LUA_NOOBJECT`. Formal parameters are the first local variables.

The function `lua_setlocal` sets the local variable `local_number` to the value previously pushed on the stack (see Section 5.2). If the function succeeds, then it returns 1. If `local_number` is greater than the number of active local variables, or if the activation record has no debug information, then this function fails and returns 0.

7.3 Hooks

The Lua interpreter offers two hooks for debugging purposes:

```
typedef void (*lua_CHFunction) (lua_Function func, char *file, int line);
lua_CHFunction lua_setcallhook (lua_CHFunction func);

typedef void (*lua_LHFunction) (int line);
lua_LHFunction lua_setlinehook (lua_LHFunction func);
```

The first hook is called whenever the interpreter enters or leaves a function. When entering a function, its parameters are a handle to the function activation record, plus the file and the line where the function is defined (the same information which is provided by `lua_funcinfo`); when leaving a function, `func` is `LUA_NOOBJECT`, `file` is `"(return)"`, and `line` is 0.

The other hook is called every time the interpreter changes the line of code it is executing. Its only parameter is the line number (the same information which is provided by the call `lua_currentline(lua_stackedfunction(0))`). This second hook is called only if the active function has been compiled with debug information (see Section 4.9).

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

7.4 The Reflexive Debugger Interface

The library `ldblib` provides the functionallity of the debugger interface to Lua programs. If you want to use this library, your host application must open it, 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.

• `funcinfo (function)`

This function returns a table with information about the given function. The table contains the following fields:

kind : may be `"C"`, if this is a C function, `"chunk"`, if this is the main part of a chunk, or `"Lua"` if this is a Lua function.

source the source where the function was defined. 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.

def_line the line where the function was defined in the source (only valid if this is a Lua function).

where can be `"global"` if this function has a global name, or `"tag-method"` if this function is a tag method handler.

name if `where` = `global`, `name` is the global name of the function; if `where` = `tag-method`, `name` is the event name of the tag method.

• `getstack (index)`

This function returns a table with informations about the function running at level `index` of the stack. Index 0 is the current function (`getstack` itself). If `index` is bigger than the number of active functions, the function returns **nil**. The table contains all the fields returned by `funcinfo`, plus the following:

func the function at that level.

current the current line on the function execution; this will be available only when the function is precompiled with debug information.

- `getlocal (index [, local])`

This function returns information about the local variables of the function at level `index` of the stack. It can be called in three ways. When called without a `local` argument, it returns a table, which associates variable names to their values. When called with a name (a string) as `local`, it returns the value of the local variable with that name. Finally, when called with an index (a number), it returns the value and the name of the local variable with that index. (The first parameter has index 1, and so on, until the last active local variable.) In that case, the function returns **nil** if there is no local variable with the given index. The specification by index is the only way to distinguish homonym variables in a function.

- `setlocal (index, local, newvalue)`

This function changes the values of the local variables of the function at level `index` of the stack. The local variable can be specified by name or by index; see function `getlocal`.

- `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. When Lua enters a function, the hook is called with the function been called, plus the source and the line where the function is defined. When Lua exits a function, the hook is called with no arguments.

When called without arguments, this function turns off call hooks.

- `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 hook is the line number the interpreter is about to execut. This hook is called only if the active function has been compiled with debug information (see Section 4.9).

When called without arguments, this function turns off line hooks.

8 Lua Stand-alone

Although Lua has been designed as an extension language, the language can also be used as a stand-alone interpreter. An implementation of such an interpreter, called simply `lua`, is provided with the standard distribution. This program can be called with any sequence of the following arguments:

`-v` prints version information.

`-d` turns on debug information.

`-e stat` executes `stat` as a Lua chunk.

`-i` runs interactively, accepting commands from standard input until an `EOF`. Each line entered is immediately executed.

`-q` same as `-i`, but without a prompt (quiet mode).

`-` executes `stdin` as a file.

`var=value` sets global `var` with string `"value"`.

`filename` executes file `filename` as a Lua chunk.

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. For instance, an invocation like

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

will first interact with the user until an `EOF`, then will set `a` to `"test"`, and finally will run the file `prog.lua`.

When in interactive mode, a multi-line statement can be written finishing intermediate lines with a backslash (\). The prompt presented is the value of the global variable `_PROMPT`. Therefore, the prompt can be changed like below:

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

In Unix systems, Lua scripts can be made into executable programs by using the `#!` form, as in `#!/usr/local/bin/lua`.

Acknowledgments

The authors would like to thank CENPES/PETROBRAS which, jointly with TeCGraf, used extensively early versions of this system 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

Although great care has been taken to avoid incompatibilities with the previous public versions of Lua, some differences had to be introduced. Here is a list of all these incompatibilities.

Incompatibilities with version 3.1

- In the debug API, the old variables `lua_debug`, `lua_callhook` and `lua_linehook` now live inside `lua_state`. Therefore, they are no longer directly accessible, and must be manipulated only through the new functions `lua_setdebug`, `lua_setcallhook` and `lua_setlinehook`.

- Old pre-compiled code is obsolete, and must be re-compiled.

Incompatibilities with version 3.0

- To support multiple contexts, Lua 3.1 must be explicitly opened before used, with function `lua_open`. However, all standard libraries check whether Lua is already opened, so any existing program that opens at least one standard library before calling Lua does not need to be modified.

- Function `dostring` no longer accepts an optional second argument, with a temporary error handler. This facility is now provided by function `call`.
- Function `gsub` no longer accepts an optional fourth argument (a callback data, a table). Closures replace this feature with advantage.
- The syntax for function declaration is now more restricted; for instance, the old syntax `function f[exp] (x) ... end` is not accepted in Lua 3.1. In these cases, programs should use an explicit assignment instead, such as `f[exp] = function (x) ... end`.
- Old pre-compiled code is obsolete, and must be re-compiled.
- The option `a=b` in Lua stand-alone now sets `a` to the *string* `b`, and not to the value of `b`.

Index