Reference Manual of the Programming Language Lua 2.1

Roberto Ierusalimschy Luiz Henrique de Figueiredo Waldemar Celes Filho

TeCGraf — PUC-Rio

roberto,lhf,celes@icad.puc-rio.br

February 8, 1995

Abstract

Lua is an extension programming language designed to be used as a configuration language for any program that needs one. This document describes version 2.1 of the Lua programming language and the API that allows interaction between Lua programs and its host C program. It also presents some examples of using the main features of the system.

Sumário

Lua é uma linguagem de extensão projetada para ser usada como linguagem de configuração em qualquer programa que precise de uma. Este documento descreve a versão 2.1 da linguagem de programação Lua e a Interface de Programação que permite a interação entre programas Lua e o programa C hospedeiro. O documento também apresenta alguns exemplos de uso das principais características do sistema.

1 Introduction

Lua is an extension programming language designed to support general procedural programming features with data description facilities. It is supposed to be used as a configuration language for any program that needs one. Its main extensions are related to object-oriented facilities, and fallbacks, but it has some other minor contributions. Lua has been designed and implemented by W. Celes F., L. H. de Figueiredo and R. Ierusalimschy.

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 rather different domains, thus creating customized programming languages sharing a syntactical framework.

Lua is free distribution software, and provided as usual with no guarantees. The implementation described in this manual is available by anonymous ftp from

`ftp.icad.puc-rio.br:/pub/lua/lua-2.1.tar.Z`

or by WWW (World Wide Web) from

`http://www.inf.puc-rio.br/projetos/roberto/lua.html`

2 Environment and Modules

All statements in Lua are executed in a *global environment*. This environment, which keeps all global variables and functions, is initialized at the beginning of the embedding program and persists until its end.

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

Global variables do not need declaration. Any variable is assumed to be global unless explicitly declared local (see local declarations, Section 4.4.5). Before the first assignment, the value of a global variable is **nil**.

The unit of execution of Lua is called a *module*. The syntax for modules is:[1]

$$module \quad \rightarrow \quad \{statement \mid function\}$$

A module may contain statements and function definitions, and may be in a file or in a string inside the host program. When a module is executed, first all its functions and statements are compiled, and the functions added to the global environment; then the statements are executed in sequential order. All modifications a module effects on the global environment persist after its end. Those include modifications to global variables and definitions of new functions[2].

3 Types

Lua is a dynamically typed language. Variables do not have types; only values do. All values carry their own type. Therefore, there are no type definitions in the language.

There are seven basic types in Lua: *nil*, *number*, *string*, *function*, *CFunction*, *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 (floating point) numbers, while *string* has the usual meaning.

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. When a function is defined in Lua, its body is compiled and stored in a given variable. Lua can call (and manipulate) functions written in Lua and functions written in C; the latter have type *CFunction*.

The type *userdata* is provided to allow arbitrary C pointers to be stored in Lua variables. It corresponds to `void*` and has no pre-defined operations in Lua, besides assignment and equality test. However, by using fallbacks, the programmer may define operations for *userdata* values; see Section 4.7.

The type *table* implements associative arrays, that is, arrays which 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. To represent a record, 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.

It is important to notice 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 4.5.7.

---

[1] As usual, $\{a\}$ means 0 or more a's, $[a]$ means an optional a and $\{a\}^+$ means one or more a's.

[2] Actually, a function definition is an assignment to a global variable; see Section 3.

4 The Language

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

4.1 Lexical Conventions

Lua is a case sensitive language. Identifiers can be any string of letters, digits, and underscores, not beginning with a digit. The following words are reserved, and cannot be used as identifiers:

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

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 `'\n'`, `'\t'` and `'\r'`. Comments start anywhere outside a string with a double hyphen (`--`) and run until the end of the line.

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

```
4     4.     .4     4.57e-3     .3e12
```

4.2 Coercion

Lua provides some automatic conversions. 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, according to the following rule: if the number is an integer, it is written without exponent or decimal point; otherwise, it is formatted following the "`%g`" conversion specification of the standard `printf` C function.

4.3 Adjustment

Functions in Lua can return many values. Because there are no type declarations, 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, the last values are thrown away. If there are more needs than values, the list is extended with as many **nil**'s as needed. Adjustment occurs in multiple assignment and function calls.

4.4 Statements

Lua supports an almost conventional set of statements. The conventional commands include assignment, control structures and procedure calls. Non-conventional commands include table constructors, explained in Section 4.5.7, and local variable declarations.

4.4.1 Blocks

A block is a list of statements, executed sequentially. Any statement can be optionally followed by a semicolon.

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

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" errors.

4.4.2 Assignment

The language allows multiple assignment. Therefore, the syntax 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
```

Before the assignment, the list of values is *adjusted* to the length of the list of variables; see Section 4.3.

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

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

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

Square brackets are used to index a table. If `var` results in a table value, the field indexed by the expression value gets the assigned value. Otherwise, the fallback *settable* is called, with three parameters: the value of `var`, the value of expression, and the value being assigned to it; see Section 4.7.

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

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

4.4.3 Control Structures

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

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

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

$$\textit{ret} \rightarrow \textbf{return}\ \textit{explist}$$

4.4.4 Expressions as Statements

All expressions with possible side-effects can be executed as statements. These include function calls and table constructors:

$$\begin{array}{lcl} \mathit{stat} & \rightarrow & \mathit{functioncall} \\ \mathit{stat} & \rightarrow & \mathit{tableconstructor} \end{array}$$

Eventual returned values are thrown away. Function calls are explained in Section 4.5.8; constructors are the subject of Section 4.5.7.

4.4.5 Local Declarations

Local variables can 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:

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

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

4.5 Expressions

4.5.1 Simple Expressions

Simple expressions are:

$$\begin{array}{lcl} \mathit{exp} & \rightarrow & \text{'('}\ \mathit{exp}\ \text{')'} \\ \mathit{exp} & \rightarrow & \textbf{nil} \\ \mathit{exp} & \rightarrow & \text{'number'} \\ \mathit{exp} & \rightarrow & \text{'literal'} \\ \mathit{exp} & \rightarrow & \mathit{var} \end{array}$$

Numbers (numerical constants) and string literals are explained in Section 4.1. Variables are explained in Section 4.4.2.

4.5.2 Arithmetic Operators

Lua supports the usual arithmetic operators. These operators are the binary `+`, `-`, `*`, `/` and `^` (exponentiation), and the unary `-`. If the operands are numbers, or strings that can be converted to numbers, according to the rules given in Section 4.2, all operations but exponentiation have the usual meaning. Otherwise, the fallback "arith" is called; see Section 4.7. An exponentiation always calls this fallback.

4.5.3 Relational Operators

Lua offers the following relational operators:

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

All return **nil** as false and 1 as true.

Equality first compares the types of its operands. If they are different, the result is **nil**. Otherwise, their values are compared. Numbers and strings are compared in the usual way.

Tables, CFunctions, 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 (`=`).

The other operators work as follows. If both arguments are numbers, they are compared as such. Otherwise, if both arguments can be converted to strings, their values are compared using lexicographical order. Otherwise, the fallback "order" is called; see Section 4.7.

4.5.4 Logical Operators

All logical operators, like control structures, consider **nil** as false and anything else as true. Like relational operators, they return **nil** as false and 1 as true. The logical operators are:

```
and   or   not
```

The operators `and` and `or` use short-cut evaluation, that is, the second operand is evaluated only if necessary.

4.5.5 Concatenation

Lua offers a string concatenation operator, denoted by "`..`". If operands are strings or numbers, they are converted to strings according to the rules in Section 4.2. Otherwise, the fallback "concat" is called; see Section 4.7.

4.5.6 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 `^`, which is right associative.

4.5.7 Table Constructors

Table constructors are expressions that create tables. They can be used to create empty tables, or to create a table and initialize some fields.

The general syntax for constructors is:

tableconstructor → '{' [*fieldlist*] '}'
fieldlist → *lfieldlist1* [',']
fieldlist → *ffieldlist1* [',']
fieldlist → *lfieldlist1* ';' *ffieldlist1* [',']

The form *lfieldlist1* is used to initialize lists.

lfieldlist1 → *exp* {',' *exp*}

The expressions in the list are assigned to consecutive numerical indexes, starting with 1. As an example:

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

is equivalent to:

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

The next form initializes named fields in a table.

ffieldlist1 → *ffield* {',' *ffield*}
ffield → *name* '=' *exp*

As an example:

```
a = {x = 1, y = 3}
```

is equivalent to:

```
temp = {}
temp.x = 1
temp.y = 3
a = temp
```

4.5.8 Function Calls

A function call has the following syntax:

functioncall → *var realParams*

Here, `var` can be any variable (global, local, indexed, etc). If its type is *function* or *CFunction*, this function is called. Otherwise, the fallback "function" is called, having as first parameter the value of `var`, and then the original call parameters.

The form:

functioncall → *var* ':' *name realParams*

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

```
var.name(var, ...)
```

except that `var` is evaluated only once.

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

All argument expressions are evaluated before the call; then the list of arguments is adjusted to the length of the list of parameters (see Section 4.3); finally, this list is assigned to the formal parameters. A call of the form `f{...}` is syntactic sugar for `f({...})`, that is, the parameter list is a single new table.

Because a function can return any number of results (see Section 4.4.3), the number of results must be adjusted before used. If the function is called as an statement (see Section 4.4.4), its return list is adjusted to 0. If the function is called in a place that needs a single value (syntactically denoted by the non-terminal `exp1`), its return list is adjusted to 1. If the function is called in a place that can hold many values (syntactically denoted by the non-terminal `exp`), no adjustment is done.

4.6 Function Definitions

Functions in Lua can be defined anywhere in the global level of a module. The syntax for function definition is:

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

When Lua finds a function definition, its body is compiled to intermediate code and stored, with type *function*, into the global variable `name`.

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

parlist1 → *'name'* {',' *name*}

Results are returned using the `return` statement (see Section 4.4.3). If control reaches the end of a function without a return instruction, the function returns with no results.

There is a special syntax for definition of methods, that is, functions which are to be stored in table fields.

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

A declaration like

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

is equivalent to

```
function temp (self, ...)
  ...
end
t.f = temp
```

that is, the function is created with a dummy name and then assigned to the field `f` of the table `t`. Moreover, the function gets an extra formal parameter called `self`. Notice that the variable `t` must be previously initialized with a table value.

4.7 Fallbacks

Lua provides a powerful mechanism to extend its semantics, called *fallbacks*. Basically, a fallback is a programmer defined function which is called whenever Lua does not know how to proceed.

Lua supports the following fallbacks, identified by the given strings:

"arith" called when an arithmetic operation is applied to non numerical operands, or when the binary ^ operation is called. Receives three arguments: the two operands (the second one is nil when the operation is unary minus) and one of the following strings describing the offended operator:

```
add  sub  mul  div  pow  unm
```

Its return value is the final result of the arithmetic operation. The default function issues an error.

"order" called when an order comparison is applied to non numerical or non string operands. Receives three arguments: the two operands and one of the following strings describing the offended operator:

```
lt gt le ge
```

Its return value is the final result of the comparison operation. The default function issues an error.

"concat" called when a concatenation is applied to non string operands. Receives the two operands as arguments. Its return value is the final result of the concatenation operation. The default function issues an error.

"index" called when Lua tries to retrieve the value of an index not present in a table. Receives as arguments the table and the index. Its return value is the final result of the indexing operation. The default function returns nil.

"gettable" called when Lua tries to index a non table value. Receives as arguments the non table value and the index. Its return value is the final result of the indexing operation. The default function issues an error.

"settable" called when Lua tries to assign indexed a non table value. Receives as arguments the non table value, the index, and the assigned value. The default function issues an error.

"function" called when Lua tries to call a non function value. Receives as arguments the non function value and the arguments given in the original call. Its return values are the final results of the call operation. The default function issues an error.

"gc" called during garbage collection. Receives as argument the table being collected. After each run of the collector this function is called with argument nil. Because this function operates during garbage collection, it must be used with great care, and programmers should avoid the creation of new objects (tables or strings) in this function. The default function does nothing.

"error" called when an error occurs. Receives as argument a string describing the error. The default function prints the message on the standard error output.

The function `setfallback` is used to change a fallback action. Its first argument is a string describing the fallback, and the second the new function to be called. It returns the old function for the given fallback, or nil on error.

Section 7.4 shows an example of the use of fallbacks.

4.8 Error Handling

Because Lua is an extension language, all Lua actions start from C code calling a function from the Lua library. Whenever an error occurs during Lua compilation or execution, an error fallback function is called, and then the correspondent function from the library (`lua_dofile`, `lua_dostring`, `lua_call`, and `lua_callfunction`) is terminated returning an error condition.

The only argument to the error fallback function is a string describing the error and some extra informations, like current line (when the error is at compilation) or current function (when the error is at execution). For more information about an error, the Lua program can include the compilation pragma `$debug`. This pragma must be written in a line by itself. When an error occurs in a program compiled with this option, the error message includes extra information showing the stack of calls.

The standard error routine only prints the error message to `stderr`. If needed, it is possible to change the error fallback routine; see Section 4.7.

Lua code can generate an error by calling the function `error`. Its optional parameter is a string, which is used as the error message.

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

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

All API functions are declared in the file `lua.h`.

5.1 Executing Lua Code

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

```
int            lua_dofile              (char *filename);
int            lua_dostring            (char *string);
```

Both functions return an error code: 0, in case of success; non zero, in case of errors.

5.2 Converting 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.

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

When C code calls Lua repeatedly, as in a loop, objects returned by there calls may acumulate, creating a memory problem. 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.

To check the type of a `lua_Object`, the following function is available:

```
int             lua_type                  (lua_Object object);
```

plus the following macros:

```
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_iscfunction           (lua_Object object);
int             lua_isuserdata            (lua_Object object);
```

All macros return 1 if the object has the given type, and 0 otherwise.

The function `lua_type` can be used to distinguish between different kinds of user data; see below.

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);
lua_CFunction   lua_getcfunction          (lua_Object object);
void           *lua_getuserdata           (lua_Object object);
```

`lua_getnumber` converts a `lua_Object` to a float. This `lua_Object` must be a number or a string convertible to number (see Section 4.2); otherwise, the function 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. Because Lua has garbage collection, there is no guarantee that such pointer will be valid after the block ends.

`lua_getcfunction` converts a `lua_Object` to a C function. This `lua_Object` must have type *CFunction*; otherwise, the function returns 0 (the null pointer). The type `lua_CFunction` is explained in Section 5.5.

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

The reverse process, that is, passing a specific C value to Lua, is done by using the following functions:

```
void            lua_pushnumber            (double n);
void            lua_pushstring            (char *s);
void            lua_pushliteral           (char *s);
void            lua_pushcfunction         (lua_CFunction f);
void            lua_pushusertag           (void *u, int tag);
```

plus the macro:

```
void            lua_pushuserdata          (void *u);
```

All of them receive a C value, convert it to a `lua_Object`, and leave their results on the top of the Lua stack, where it can be assigned to a variable, passed as paramenter to a Lua function, etc (see below). `lua_pushliteral` is like `lua_pushstring`, but also puts the string in the Lua literal table. This avoids the string to be garbage collected, and therefore has a better overall performance. As a rule, when the string to be pushed is a literal, `lua_pushliteral` should be used.

User data can have different tags, whose semantics are defined by the host program. Any positive integer can be used to tag a user data. When a user data is retrieved, the function `lua_type` can be used to get its tag.

To complete the set, the value **nil** or a `lua_Object` can also be pushed onto the stack, with:

```
void            lua_pushnil            (void);
void            lua_pushobject         (lua_Object object);
```

5.3 Manipulating Lua Objects

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

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

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

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

Tables can also be manipulated via the API. The function

```
lua_Object      lua_getsubscript       (void);
```

expects on the stack a table and an index, and returns the contents of the table at that index. As in Lua, if the first object is not a table, or the index is not present in the table, the correspondent fallback is called.

For compatibility with previous versions of the API, the following macros are supported:

```
lua_Object      lua_getindexed         (lua_Object table, float index);
lua_Object      lua_getfield           (lua_Object table, char *field);
```

The first one is used for numeric indices, while the second can be used for any string index.

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

```
void lua_storesubscript (void);
```

Again, the correspondent fallback is called if needed.

Finally, the function

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

creates a new table.

Please Notice: Most functions from the Lua library receive parameters through the stack. Because other functions also use the stack, it is important that these parameters be pushed just before the correspondent call, without intermediate calls to the Lua library. For instance, suppose the user wants the value of `a[i]`. A simplistic solution would be:

```
/* Warning: WRONG CODE */
lua_Object result;
lua_pushobject(lua_getglobal("a"));  /* push table */
lua_pushobject(lua_getglobal("i"));  /* push index */
result = lua_getsubscript();
```

However, the call `lua_getglobal("i")` modifies the stack, and invalidates the previous pushed value. A correct solution could be:

```
lua_Object index, result;
index = lua_getglobal("i");
lua_pushobject(lua_getglobal("a"));  /* push table */
lua_pushobject(index);               /* push index */
result = lua_getsubscript();
```

5.4 Calling Lua Functions

Functions defined in Lua by a module executed with `dofile` or `dostring` can be called from the host program. This is done using the following protocol: first, the arguments to the function are pushed onto the Lua stack (see Section 5.2), in direct order, i.e., the first argument is pushed first. Again, it is important to emphasize that, during this phase, no other Lua function can be called.

Then, the function is called using

```
int             lua_call                 (char *functionname);
```

or

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

Both functions return an error code: 0, in case of success; non zero, in case of errors. Finally, the returned values (a Lua function may return many values) can be retrieved with the macro

```
lua_Object      lua_getresult             (int number);
```

`number` is the order of the result, starting with 1. When called with a number larger than the actual number of results, this function returns `LUA_NOOBJECT`.

Two special Lua functions have exclusive interfaces: `error` and `setfallback`. 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, the corresponding Lua execution terminates, as if an error had occured inside Lua code. Otherwise, the whole program terminates.

Fallbacks can be changed with:

```
lua_Object lua_setfallback (char *name, lua_CFunction fallback);
```

The first parameter is the fallback name, and the second a CFunction to be used as the new fallback. This function returns a `lua_Object`, which is the old fallback value, or nil on fail (invalid fallback name). This old value can be used for chaining fallbacks.

An example of C code calling a Lua function is shown in Section 7.6.

5.5 C Functions

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

```
#define lua_register(n,f)       (lua_pushcfunction(f), lua_storeglobal(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.

To access its arguments, a C function calls:

```
lua_Object     lua_getparam             (int number);
```

`number` starts with 1 to get the first argument. When called with a number larger than the actual number of arguments, this function returns `LUA_NOOBJECT`. In this way, it is possible to write functions that work with a variable number of parameters.

To return values, a C function just pushes them onto the stack, in direct order; see Section 5.2. Like a Lua function, a C function called by Lua can also return many results.

Section 7.5 presents an example of a CFunction.

5.6 Locking Lua Objects

As already noted, `lua_Object`s are volatile. If the C code needs to keep a `lua_Object` outside block boundaries, it has to *lock* the object. The routines to manipulate locking are the following:

```
int        lua_lock (void);
lua_Object lua_getlocked  (int ref);
void       lua_pushlocked (int ref);
void       lua_unlock (int ref);
```

The function `lua_lock` locks the object which are on the top of the stack, and returns a reference to it. Whenever the locked object is needed, a call to `lua_getlocked` returns a handle to it, while `lua_pushlocked` pushes the handle on the stack. When a locked object is no longer needed, it can be unlocked with a call to `lua_unlock`.

6 Predefined Functions and Libraries

The set of predefined functions in Lua is small but powerful. Most of them provide features that allows some degree of reflexivity in the language. Many of these features cannot be simulated with the rest of the Language nor with the standard API.

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 separated C modules. Currently there are three standard libraries:

- string manipulation;
- mathematical functions (sin, cos, etc);
- input and output;

6.1 Predefined Functions

`dofile (filename)`

This function receives a file name, opens it and executes its contents as a Lua module. It returns 1 if there are no errors, **nil** otherwise.

`dostring (string)`

This function executes a given string as a Lua module. It returns 1 if there are no errors, **nil** otherwise.

`next (table, index)`

This function allows a program to enumerate all fields of a table. Its first argument is a table and its second argument is an index in this table; this index can be a number or a string. 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**.

In Lua there is 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 the indices are enumerated are not specified, *even for numeric indices*.

See Section 7.1 for an example of the use of this function.

`nextvar (name)`

This function is similar to the function `next`, but it iterates 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. See Section 7.1 for an example of the use of this function.

`print (e1, e2, ...)`

This function receives any number of arguments, and prints their values in a reasonable format. Each value is printed in a new line. This function is not intended for formatted output, but as a quick way to show a value, for instance for error messages or debugging. See Section 6.4 for functions for formatted output.

`tonumber (e)`

This function 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 (see Section 4.2), it returns that number; otherwise, it returns **nil**.

`type (v)`

This function 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'`, `'number'`, `'string'`, `'table'`, `'cfunction'`, `'function'`, and `'userdata'`.

`error (message)`

This function issues an error message and terminates the last called function from the library (`lua_dofile`, `lua_dostring`, ...). It never returns.

`setglobal (name, value)`

This function 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`.

`getglobal (name)`

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

`setfallback (fallbackname, newfallback)`

This function sets a new fallback function to the given fallback. It returns the old fallback function, or nil in case of error (invalid fallback name, or `newfallback` is not a function).

6.2 String Manipulation

This library provides generic functions for string manipulation, such as finding and extracting substrings. When indexing a string, the first character has position 1. See Section 7.2 for some examples on string manipulation in Lua.

`strfind (str, substr, [init, [end]])`

Receives two string arguments, and returns a number. This number indicates the first position where the second argument appears in the first argument. If the second argument is not a substring of the first one, then `strfind` returns **nil**. A third optional numerical argument specifies where to start the search. Another optional numerical argument specifies where to stop it.

`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 runing until `j`. If `j` is absent or is **nil**, it is assumed to be equal to the length of `s`. Particularly, the call `strsub(s,1,j)` returns a prefix of `s` with length `j`, while the call `strsub(s,i)` returns a suffix of `s`, starting at `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.

`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.

6.3 Mathematical Functions

This library is an interface to some functions of the standard C math library. Moreover, it registers a fallback for the binary operator `^` which, when applied to numbers `x^y`, returns x^y.

The library provides the following functions:

```
abs acos asin atan ceil cos floor log log10
max min  mod  sin  sqrt tan
```

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.

The function `max` returns the maximum value in a list of numeric arguments. Similarly, `min` computes the minimum. Both can be used with an unlimited number of arguments.

The function `mod` is equivalent to the `%` operator in C.

6.4 I/O Facilities

All I/O operations in Lua are done over two *current* files, one for reading and one for writing. Initially, the current input file is `stdin`, and the current output file is `stdout`.

Unless otherwised stated, all I/O functions return 1 on success and **nil** on failure.

`readfrom (filename)`

This function opens a file named `filename` and sets it as the *current* input file. When called without parameters, this function restores `stdin` as the current input file.

`writeto (filename)`

This function opens a file named `filename` and sets it as the *current* output file. Notice that, if the file already exists, it is completely erased with this operation. When called without parameters, this function restores `stdout` as the current output file.

`appendto (filename)`

This function opens a file named `filename` and sets it as the *current* output file. Unlike the `writeto` operation, this function does not erase any previous content of the file. When called without parameters, this function restores `stdout` as the current output file. This function returns 2 if the file already exists, 1 if it creates a new file, and **nil** on failure.

`read ([format])`

This function returns a value read from the current input. An optional string argument specifies the way the input is interpreted.

Without a format argument, `read` first skips blanks, tabs and newlines. Then it checks whether the current character is `"` or `'`. If so, it reads a string up to the ending quotation mark, and returns this string, without the quotation marks. Otherwise it reads up to a blank, tab or newline.

The format string can have the following format:

`?[n]`

where `?` can be:

's' or 'S' to read a string;

'f' or 'F' to read a real number;

'i' or 'I' to read an integer.

The optional `n` is a number which specifies how many characters must be read to compose the input value.

`write (value, [format])`

This function writes the value of its first argument to the current output. An optional second argument specifies the format to be used. This format is given as a string, composed of four parts. The first part is the only not optional, and must be one of the following characters:

's' or 'S' to write strings;

'f' or 'F' to write floats;

'i' or 'I' to write integers.

These characters can be followed by

`[?][m][.n]`

where:

`?` indicates justification inside the field.

- '`<`' right justification;
- '`>`' left justification;
- '`|`' center justification.

`m` Indicates the field size in characters.

`.n` For reals, indicates the number of digital places. For integers, it is the minimum number of digits. This option has no meaning for strings.

When called without a format string, this function writes numbers using the `%g` format and strings with `%s`.

`debug ()`

This function, when called, repeatedly presents a prompt `lua_debug>` in the error output stream (`stderr`), reads a line from the standard input, and executes ("dostring") the line. The loop ends when the user types `cont` to the prompt. This function then returns and the execution of the program continues.

7 Some Examples

This section gives examples showing some features of Lua. It does not intend to cover the whole language, but only to illustrate some interesting uses of the system.

7.1 The Functions `next` and `nextvar`

This example shows how to use the function **`next`** to iterate over the fields of a table.

```
function f (t)                  -- t is a table
  local i, v = next(t, nil)   -- i is an index of t, v = t[i]
  while i do
    ...                         -- do something with i and v
    i, v = next(t, i)           -- get next index
  end
end
```

The next example prints the names of all global variables in the system with non nil values:

```
function printGlobalVariables ()
  local i, v = nextvar(nil)
  while i do
    print(i)
    i, v = nextvar(i)
  end
end
```

7.2 String Manipulation

The first example is a function to trim extra blanks at the beginning and end of a string.

```
function trim(s)
  local l = 1
  while strsub(s,l,l) == ' ' do
    l = l+1
  end
  local r = strlen(s)
  while strsub(s,r,r) == ' ' do
    r = r-1
  end
  return strsub(s,l,r)
end
```

The second example shows a function that eliminates all blanks of a string.

```
function remove_blanks (s)
  local b = strfind(s, ' ')
  while b do
    s = strsub(s, 1, b-1) .. strsub(s, b+1)
    b = strfind(s, ' ')
  end
  return s
end
```

7.3 Persistence

Because of its reflexive facilities, persistence in Lua can be achieved with Lua. This section shows some ways to store and retrieve values in Lua, using a text file written in the language itself as the storage media.

To store a single value with a name, the following code is enough:

```
function store (name, value)
  write('\n' .. name .. '=')
  write_value(value)
end

function write_value (value)
  local t = type(value)
      if t == 'nil'    then write('nil')
  elseif t == 'number' then write(value)
  elseif t == 'string' then write('"' .. value .. '"')
  end
end
```

In order to restore this value, a `lua_dofile` suffices.

Storing tables is a little more complex. Assuming that the table is a tree, and all indices are identifiers (that is, the tables are being used as records), its value can be written directly with table constructors. First, the function `write_value` is changed to

```
function write_value (value)
  local t = type(value)
      if t == 'nil'    then write('nil')
  elseif t == 'number' then write(value)
  elseif t == 'string' then write('"' .. value .. '"')
  elseif t == 'table'  then write_record(value)
  end
end
```

The function `write_record` is:

```
function write_record(t)
  local i, v = next(t, nil)
  write('{')  -- starts constructor
  while i do
    store(i, v)
    write(', ')
    i, v = next(t, i)
  end
  write('}')  -- closes constructor
end
```

7.4 Inheritance

The fallback for absent indices can be used to implement many kinds of inheritance in Lua. As an example, the following code implements single inheritance:

```
function Index (t,f)
  if f == 'parent' then  -- to avoid loop
    return OldIndex(t,f)
  end
  local p = t.parent
  if type(p) == 'table' then
    return p[f]
  else
    return OldIndex(t,f)
  end
end

OldIndex = setfallback("index", Index)
```

Whenever Lua attempts to access an absent field in a table, it calls the fallback function `Index`. If the table has a field `parent` with a table value, then Lua attempts to access the desired field in this parent object. This process is repeated "upwards" until a value for the field is found or the object has no parent. In the latter case, the previous fallback is called to supply a value for the field.

When better performance is needed, the same fallback may be implemented in C, as ilustrated in Figure 1. This code must be registered with:

```
  lua_pushliteral("parent");
  lockedParentName = lua_lock();
  lua_pushobject(lua_setfallback("index", Index));
  lockedOldIndex = lua_lock();
```

Notice that the string `"parent"` is kept locked in Lua for optimal performace.

7.5 A CFunction

A CFunction to compute the maximum of a variable number of arguments is shown in Figure 2. After registered with

```
lua_register ("max", math_max);
```

this function is available in Lua, as follows:

```
i = max(4, 5, 10, -34)  -- i receives 10
```

7.6 Calling Lua Functions

This example illustrates how a C function can call the Lua function `remove_blanks` presented in Section 7.2.

```
void remove_blanks (char *s)
{
  lua_pushstring(s);  /* prepare parameter */
  lua_call("remove_blanks");  /* call Lua function */
  strcpy(s, lua_getstring(lua_getresult(1)));  /* copy result back to 's' */
}
```

```c
int lockedParentName;  /* stores the lock index for the string "parent" */
int lockedOldIndex;    /* previous fallback function */

void callOldFallback (lua_Object table, lua_Object index)
{
  lua_Object oldIndex = lua_getlocked(lockedOldIndex);
  lua_pushobject(table);
  lua_pushobject(index);
  lua_callfunction(oldIndex);
}

void Index (void)
{
  lua_Object table = lua_getparam(1);
  lua_Object index = lua_getparam(2);
  lua_Object parent;
  if (lua_isstring(index) && strcmp(lua_getstring(index), "parent") == 0)
  {
    callOldFallback(table, index);
    return;
  }
  lua_pushobject(table);
  lua_pushlocked(lockedParentName);
  parent = lua_getsubscript();
  if (lua_istable(parent))
  {
    lua_pushobject(parent);
    lua_pushobject(index);
    /* return result from getsubscript */
    lua_pushobject(lua_getsubscript());
  }
  else
    callOldFallback(table, index);
}
```

Figure 1: Inheritance in C.

```
void math_max (void)
{
 int i=1;   /* number of arguments */
 double d, dmax;
 lua_Object o;
 /* the function must get at least one argument */
 if ((o = lua_getparam(i++)) == LUA_NOOBJECT)
   lua_error ("too few arguments to function `max'");
 /* and this argument must be a number */
 if (!lua_isnumber(o))
   lua_error ("incorrect argument to function `max'");
 dmax = lua_getnumber (o);
 /* loops until there is no more arguments */
 while ((o = lua_getparam(i++)) != LUA_NOOBJECT)
 {
  if (!lua_isnumber(o))
    lua_error ("incorrect argument to function `max'");
  d = lua_getnumber (o);
  if (d > dmax) dmax = d;
 }
 /* push the result to be returned */
 lua_pushnumber (dmax);
}
```

Figure 2: C function `math_max`.

Acknowledgments

The authors would like to thank CENPES/PETROBRÁS 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[3].

A 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 differences.

Incompatibilities in the Language

- The equality test operator now is denoted by `==`, instead of `=`.
- The syntax for table construction has been greatly simplified. The old `@(size)` has been substituted by `{}`. The list constructor (formerly `@[...]`) and the record constructor (formerly `@{...}`) now are both coded like `{...}`. When the construction involves a function call, like in `@func{...}`, the new syntax does not use the `@`. More important, *a construction function must now explicitly return the constructed table.*

Incompatibilities in the API

- The function `lua_call` no longer has the parameter `nparam`.
- The function `lua_pop` is no longer available, since it could lead to strange behavior. In particular, to access results returned from a Lua function, the new macro `lua_getresult` should be used.
- The old functions `lua_storefield` and `lua_storeindexed` have been replaced by

  ```
  int lua_storesubscript (void);
  ```

  with the parameters explicitly pushed on the stack.
- The functionality of the function `lua_errorfunction` has been replaced by the *fallback* mechanism; see Section 4.8.
- When calling a function from the Lua library, parameters passed through the stack must be pushed just before the correspondent call, with no intermediate calls to Lua. Special care should be taken with macros like `lua_getindexed` and `lua_getfield`.

[3] BTW, Lua means *moon* in Portuguese.