Reference Manual of the Programming Language Lua 2.5

Roberto Ierusalimschy   Luiz Henrique de Figueiredo   Waldemar Celes

lua@icad.puc-rio.br

TeCGraf — Departamento de Informática — PUC-Rio

Date: 1996/11/18 14:27:42

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 2.5 of the Lua programming language and the API that allows interaction between Lua programs and their host C programs. The document 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 2.5 da linguagem de programação Lua e a Interface de Programação (API) que permite a interação entre programas Lua e programas C hospedeiros. 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 intended to be used as a configuration language for any program that needs one. Lua has been designed and implemented by W. Celes, 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 at the following URL's:

```
http://www.inf.puc-rio.br/~roberto/lua.html
ftp://ftp.icad.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 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 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 *chunk*. The syntax[1] for chunks is:

$$chunk \quad \rightarrow \quad \{statement \mid function\}\ [ret]$$

A chunk may contain statements and function definitions, and may be in a file or in a string inside the host program. A chunk may optionally ends with a return statement (see Section 4.4.3). When a chunk is executed, first all its functions and statements are compiled, then the statements are executed in sequential order. All modifications a chunk effects on the global environment persist after its end. Those include modifications to global variables and definitions of new functions[2].

Chunks may be pre-compiled; 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

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 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. 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.

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

[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.

to tables, and do not imply any kind of copy. Moreover, tables must be explicitly created before used (see Section 4.5.7).

4 The Language

This section describes the lexis, the syntax and the 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    then      until     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'`. 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.

Comments start anywhere outside a string with a double hyphen (`--`) and run until the end of the line. Moreover, if the first line of a chunk file starts with `#`, this line is skipped[3].

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

```
4     4.0     0.4     4.57e-3     0.3e12
```

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 `printf` function in the standard C library.

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, then the last 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 and function calls.

[3]This facility allows the use of Lua as a script interpreter in Unix systems.

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 (Section 4.5.7), and local variable declarations (Section 4.4.5).

4.4.1 Blocks

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

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

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:

$$
\begin{aligned}
\mathit{stat} &\rightarrow \mathit{varlist1}\ \text{'='}\ \mathit{explist1} \\
\mathit{varlist1} &\rightarrow \mathit{var}\ \{\text{','}\ \mathit{var}\}
\end{aligned}
$$

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

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

$$\mathit{var} \rightarrow \mathit{name}$$

Square brackets are used to index a table:

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

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.

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

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

4.4.3 Control Structures

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

$$
\begin{aligned}
\mathit{stat} &\rightarrow \textbf{while}\ \mathit{exp1}\ \textbf{do}\ \mathit{block}\ \textbf{end} \\
&\mid\ \textbf{repeat}\ \mathit{block}\ \textbf{until}\ \mathit{exp1} \\
&\mid\ \textbf{if}\ \mathit{exp1}\ \textbf{then}\ \mathit{block}\ \{\mathit{elseif}\}\ [\textbf{else}\ \mathit{block}]\ \textbf{end} \\
\mathit{elseif} &\rightarrow \textbf{elseif}\ \mathit{exp1}\ \textbf{then}\ \mathit{block}
\end{aligned}
$$

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

$$\mathit{ret} \;\rightarrow\; \textbf{return}\ \mathit{explist}\ [\mathit{sc}]$$

4.4.4 Function Calls as Statements

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

$$\mathit{stat} \;\rightarrow\; \mathit{functioncall}$$

Eventual returned values are thrown away. Function calls are explained in Section 4.5.8.

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 present, an initial assignment 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:

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

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, then all operations but exponentiation have the usual meaning. Otherwise, the fallback "arith" is called (see Section 4.7). An exponentiation always calls this fallback. The standard mathematical library redefines this fallback, giving the expected meaning to exponentiation (see Section 6.3).

4.5.3 Relational Operators

Lua provides the following relational operators:

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

All these return **nil** as false and a value different from **nil** (actually the number 1) as true.

Equality first compares the types 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, 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, then they are compared as such. Otherwise, if both arguments can be converted to strings, their values are compared using lexicographical order. Otherwise, the "order" fallback is called (see Section 4.7).

4.5.4 Logical Operators

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

```
and   or   not
```

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 if necessary.

4.5.5 Concatenation

Lua offers a string concatenation operator, denoted by "`..`". If operands are strings or numbers, then 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 `^` (exponentiation), which is right associative.

4.5.7 Table Constructors

Table constructors are expressions that create tables; every time a constructor is evaluated, a new table is created. Constructors can be used to create empty tables, or to create a table and initialize some fields.

The general syntax for constructors is:

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

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 indexes, starting with 1. For example:

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

is roughly equivalent to:

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

The next form initializes named fields in a table:

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

For example:

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

is roughly equivalent to:

```
temp = {}
temp.x = 1    -- or temp["x"] = 1
temp.y = 3    -- or temp["y"] = 3
a = temp
```

4.5.8 Function Calls

A function call has the following syntax:

$$\textit{functioncall} \rightarrow \textit{var realParams}$$

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

The form:

$$\textit{functioncall} \rightarrow \textit{var}\ \text{':'}\ \textit{name realParams}$$

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

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

except that `var` is evaluated only once.

$$\begin{array}{rcl} \textit{realParams} & \rightarrow & \text{'('}\ [\textit{explist1}]\ \text{')'} \\ \textit{realParams} & \rightarrow & \textit{tableconstructor} \\ \textit{explist1} & \rightarrow & \textit{exp1}\ \{\text{','}\ \textit{exp1}\} \end{array}$$

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 a 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`), then 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`), then no adjustment is made.

4.6 Function Definitions

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

$$\textit{function} \rightarrow \textbf{function}\ \textit{var}\ \text{'('}\ [\textit{parlist1}]\ \text{')'}\ \textit{block}\ \textbf{end}$$

When Lua pre-compiles a chunk, all its function bodies are pre-compiled, too. Then, when Lua "executes" the function definition, its body is stored, with type *function*, into the variable `var`.

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

$$\textit{parlist1} \rightarrow \textit{name}\ \{\text{','}\ \textit{name}\}$$

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

There is a special syntax for defining methods, that is, functions that have an extra parameter *self*.

$$\textit{function} \rightarrow \textbf{function}\ \textit{var}\ \text{':'}\ \textit{name}\ \text{'('}\ [\textit{parlist1}]\ \text{')'}\ \textit{block}\ \textbf{end}$$

Thus, a declaration like

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

is equivalent to

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

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

4.7 Fallbacks

Lua provides a powerful mechanism to extend its semantics, called *fallbacks*. A fallback is a programmer defined function that 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. It 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 handler issues an error.

"order": called when an order comparison is applied to non numerical or non string operands. It 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 handler issues an error.

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

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

"getglobal": called when Lua tries to retrieve the value of a global variable which has a nil value (or which has not been initialized). It receives as argument the name of the variable. Its return value is the final result of the expression. The default handler returns nil.

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

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

"function": called when Lua tries to call a non function value. It 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 handler issues an error.

"gc": called during garbage collection. It 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 handler does nothing.

"error": called when an error occurs. It receives as argument a string describing the error. The default handler prints the message on the standard error output.

The function `setfallback` is used to change a fallback handler. Its first argument is the name of a fallback condition, and the second argument is the new function to be called. It returns the old handler function for the given fallback.

Section 8.6 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 corresponding function from the library (`lua_dofile`, `lua_dostring`, `lua_call`, or `lua_callfunction`) is terminated returning an error condition.

The only argument to the "error" fallback function is a string describing the error. The standard I/O library redefines this fallback, using the debug facilities (see Section 7), in order to print some extra information, like the call stack. 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 routine is able to print also the lines where the calls (and the error) were made. If needed, it is possible to change the "error" fallback handler (see Section 4.7).

Lua code can explicitly generate an error by calling the built-in function `error` (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 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. references to Lua Objects.

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

5.1 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);
```

Both 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. The function lua_dofile, if called with argument NULL (0), executes the stdin stream. Function lua_dofile is also able to execute pre-compiled chunks. It automatically detects whether the file is text or binary, and loads it accordingly (see program luac).

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. Values of type lua_Object have no meaning outside Lua; for instance, the comparisson of two lua_Object's is of no significance.

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 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 C global variables.

When C code calls Lua repeatedly, as in a loop, objects returned by these calls accumulate, and may create 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 and functions:

```
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);
```

All macros return 1 if the object is compatible with the given type, and 0 otherwise. The function lua_isnumber accepts numbers and numerical strings, lua_isstring accepts strings and numbers (see Section 4.2), and lua_isfunction accepts Lua and C functions. The function lua_type can be used to distinguish between different kinds of user data.

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_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 corresponding `lua_Object`, and leave the result on the top of the Lua stack, where it can be assigned to a Lua variable, passed as parameter to a Lua function, etc.

User data can have different tags, whose semantics are defined by the host program. Any positive integer can be used to tag a user datum. When a user datum 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 uses the function:

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

As in Lua, if the value of the global is **nil**, then the "getglobal" fallback is called.

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 corresponding fallback is called.

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 corresponding fallback is called if needed.

Finally, the function

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

creates and returns a new table.

Please Notice: Most functions from the Lua library receive parameters through Lua's stack. Because other functions also use this stack, it is important that these parameters be pushed just before the corresponding 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 result;
lua_Object index = lua_getglobal("i");
lua_pushobject(lua_getglobal("a"));  /* push table */
lua_pushobject(index);               /* push index */
result = lua_getsubscript();
```

The functions `lua_getnumber`, `lua_getstring`, `lua_getuserdata`, and `lua_getcfunction`, plus the family `lua_is*`, are safe to be called without modifying the stack.

5.4 Calling Lua Functions

Functions defined in Lua by a chunk 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);
```

where `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 occurred 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 8.10.

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);
```

where `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 8.9 presents an example of a CFunction.

5.6 References to Lua Objects

As noted in Section 5.5, `lua_Objects` are volatile. If the C code needs to keep a `lua_Object` outside block boundaries, 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_pushref (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*: that means the object will not be garbage collected. Notice that an unlocked reference may be garbage collected. Whenever the referenced object is needed, a call to `lua_getref` returns a handle to it, whereas `lua_pushref` pushes the object on the stack. If the object has been collected, then `lua_getref` returns `LUA_NOOBJECT`, and `lua_pushobject` issues an error.

When a reference is no longer needed, it can be freed 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 allows 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 separated C modules. Currently there are three standard libraries:

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

In order to have access to these libraries, the host program must call the functions `strlib_open`, `mathlib_open`, and `iolib_open`, declared in `lualib.h`.

6.1 Predefined Functions

• `dofile (filename)`

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

• `dostring (string)`

This function executes a given string as a Lua chunk. If there is any error executing the string, it returns **nil**. Otherwise, it returns the values returned by the chunk, or a non **nil** value if the chunk returns no values.

• `next (table, index)`

This function 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**.

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 is not specified, *even for numeric indices.*

See Section 8.2 for an example of the use of this function.

- `nextvar (name)`

This function is similar to the function `next`, but 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 8.2 for an example of the use of this function.

- `tostring (e)`

This function receives an argument of any type and converts it to a string in a reasonable format.

- `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), then 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"` (a string, not the value **nil**), `"number"`, `"string"`, `"table"`, `"function"` (returned both for C functions and Lua functions), and `"userdata"`.

Besides this string, the function returns a second result, which is the *tag* of the value. This tag can be used to distinguish between user data with different tags, and between C functions and Lua functions.

- `assert (v)`

This function issues an *"assertion failed!"* error when its argument is **nil**.

- `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`. It returns the value of its second argument.

• `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.

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 has position 1. See Page 19 for an explanation about patterns, and Section 8.3 for some examples on string manipulation in Lua.

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

This function looks for the first *match* of `pattern` in `str`. If it finds one, it returns the indexes on `str` where this occurence 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. A value of 1 as a forth optional argument turns off the pattern matching facilities, so the function does a plain "find substring" operation.

• `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, it is assumed to be equal to the length of `s`. In particular, the call `strsub(s,1,j)` returns a prefix of `s` with length `j`, whereas 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.

• `strrep (s, n)`

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

• `ascii (s [, i])`

Returns the ascii code of the character `s[i]`. If `i` is absent, then it is assumed to be 1.

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

This function returns a formated 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; that is, 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"
```

The options `c`, `d`, `E`, `e`, `f`, `g` `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string.

• `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, 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, this function is called every time a match occurs, with all captured substrings as parameters. If the value returned by this function is a string, it is used as the replacement string; otherwise, the replacement string is the empty string.

An 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.

As an example, in the following expression each occurrence of the form `$name$` calls the function `getenv`, passing `name` as argument (because only this part of the pattern is captured). The value returned by `getenv` will replace the pattern. Therefore, the whole expression:

```
  gsub("home = $HOME$, user = $USER$", "$(%w%w*)$", getenv)
```

may return the string:

```
home = /home/roberto, user = roberto
```

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.

`.` — represents all characters.

`%a` — represents all letters.

`%A` — represents all non letter characters.

`%d` — represents all digits.

`%D` — represents all non digits.

`%l` — represents all lower case letters.

`%L` — represents all non lower case letter characters.

`%s` — represents all space characters.

`%S` — represents all non space characters.

`%u` — represents all upper case letters.

`%U` — represents all non upper case letter characters.

`%w` — represents all alphanumeric characters.

`%W` — represents all non alphanumeric characters.

`%`x (where x is any non alphanumeric character) — represents the character x.

`[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 `-`; e.g., `A-Z` specifies the upper case characters. 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.

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

Pattern Item: a *pattern item* may be a single character class, or a character class followed by `*` or by `?`. A single character class matches any single character in the class. A character class followed by `*` matches 0 or more repetitions of characters in the class. A character class followed by `?` matches 0 or one occurrence of a character in the class. A pattern item may also has the form `%n`, for `n` between 1 and 9; such item matches a sub-string equal to the n-th captured string.

Pattern: a *pattern* is a sequence of pattern items. Any repetition item (`*`) inside a pattern will always match the longest possible sequence. 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.

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 *captured* for future use. Captures are numbered according to their left parentheses.

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 atan2 ceil cos floor log log10
max min  mod  sin  sqrt tan random randomseed
```

Most of them are only interfaces to the homonymous functions in the C library, except that, for the trigonometric functions, all angles are expressed in degrees, not radians.

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

The functions `random` and `randomseed` are interfaces to the simple random generator functions `rand` and `srand`, provided by ANSI C. The function `random` returns pseudo-random numbers in the range $[0, 1)$.

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

• `readfrom (filename)`

This function may be called in three ways. When called with a file name, it opens the named file, sets it as the *current* input file, and returns a *handle* to the file (this handle is a user data containing the file stream `FILE *`). When called with a file handle, returned by a previous call, it restores the file as the current input. When called without parameters, it closes the current input file, and restores `stdin` as the current input file.

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 open, via function `popen`.

• `writeto (filename)`

This function may be called in three ways. When called with a file name, it opens the named file, sets it as the *current* output file, and returns a *handle* to the file (this handle is a user data containing the file stream `FILE *`). Notice that, if the file already exists, it will be *completely erased* with this operation. When called with a file handle, returned by a previous call, it restores the file as the current output. When called without parameters, this function closes the current output file, and restores `stdout` as the current output file.

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 open, via function `popen`.

- `appendto (filename)`

This function opens a file named `filename` and sets it as the *current* output file. It returns the file handle, or **nil** in case of error. 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.

Notice that function `writeto` is available to close a file.

- `remove (filename)`

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

- `rename (name1, name2)`

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

- `tmpname ()`

This function returns a string with a file name that can safely be used for a temporary file.

- `read ([readpattern])`

This function reads the current input according to a read pattern, that specifies how much to read; characters are read from the current input file until the read pattern fails or ends. The function `read` returns a string with the characters read, or **nil** if the read pattern fails *and* the result string would be empty. When called without parameters, 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 `?` 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 characteres, it never fails.[4]

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.

Following are some examples of read patterns and their meanings:

- `"."` returns the next character, or **nil** on end of file.
- `".*"` reads the whole file.

[4]Notice that this behaviour is different from regular pattern matching, where a `*` expands to the maximum length *such that* the rest of the pattern does not fail.

- "[^\n]*{\n}" returns the next line (skipping the end of line), or **nil** on end of file. This is the default pattern.
- "{%s*}%S%S*" returns the next word (maximal sequence of non white-space characters), or **nil** on end of file.
- "{%s*}[+-]?%d%d*" returns the next integer or **nil** if the next characters do not conform to an integer format.

• `write (value1, ...)`

This function writes the value of each of its arguments to the current output file. The arguments must be strings or numbers. If this function fails, it returns **nil**, plus a string describing the error.

• `date ([format])`

This function 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.

• `exit ([code])`

This function calls the C function `exit`, with an optional `code`, to terminate the program.

• `getenv (varname)`

Returns the value of the 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 implementation-defined.

7 The Debugger Interface

Lua has no built-in debugger 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, a parameter `lua_Function` accepts only a handle returned by `lua_stackedfunction`.

Three other functions produce extra information about a function:

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

`lua_funcinfo` gives the file name and the line where the given function has been defined. 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 pre-compiled with debug information (see Section 4.8). When no line information is available, it returns -1.

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` first checks whether the given function is a fallback. If so, it returns the string `"fallback"`, and `name` is set to point to the fallback name. Otherwise, if the given function is the value of a global variable, then `lua_getobjname` returns the string `"global"`, while `name` points to the variable name. If the given function is neither a fallback 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 pre-compiled with debug information (see Section 4.8).

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

The first one 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`.

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);
extern lua_CHFunction lua_callhook;

typedef void (*lua_LHFunction) (int line);
extern lua_LHFunction lua_linehook;
```

The first one 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 only called if the active function has been pre-compiled with debug information (see Section 4.8).

A hook is disabled when its value is NULL (0), which is the initial value of both hooks.

8 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.

8.1 Data Structures

Tables are a strong unifying data constructor. They directly implement a multitude of data types, like ordinary arrays, records, sets, bags, and lists.

Arrays need no explanations. In Lua, it is conventional to start indices from 1, but this is only a convention. Arrays can be indexed by 0, negative numbers, or any other value (except **nil**). Records are also trivially implemented by the syntactic sugar `a.x`.

The best way to implement a set is to store its elements as indices of a table. The statement `s = {}` creates an empty set `s`. The statement `s[x] = 1` inserts the value of `x` into the set `s`. The expression `s[x]` is true if and only if `x` belongs to `s`. Finally, the statement `s[x] = nil` removes `x` from `s`.

Bags can be implemented similarly to sets, but using the value associated to an element as its counter. So, to insert an element, the following code is enough:

```
if s[x] then s[x] = s[x]+1 else s[x] = 1 end
```

and to remove an element:

```
if s[x] then s[x] = s[x]-1 end
if s[x] == 0 then s[x] = nil end
```

Lisp-like lists also have an easy implementation. The "cons" of two elements `x` and `y` can be created with the code `l = {car=x, cdr=y}`. The expression `l.car` extracts the header, while `l.cdr` extracts the tail. An alternative way is to create the list directly with `l={x,y}`, and then to extract the header with `l[1]` and the tail with `l[2]`.

8.2 The Functions `next` and `nextvar`

This example shows how to use the function `next` to iterate over the fields of a table. Function `clone` receives any table and returns a clone of it.

```
function clone (t)           -- t is a table
  local new_t = {}           -- create a new table
  local i, v = next(t, nil)  -- i is an index of t, v = t[i]
```

```
  while i do
    new_t[i] = v
    i, v = next(t, i)        -- get next index
  end
  return new_t
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
```

8.3 String Manipulation

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

```
function trim(s)
  local _, i = strfind(s, '^ *')
  local f, __ = strfind(s, ' *$')
  return strsub(s, i+1, f-1)
end
```

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

```
function remove_blanks (s)
  return gsub(s, "%s%s*", "")
end
```

8.4 Variable number of arguments

Lua does not provide any explicit mechanism to deal with variable number of arguments in function calls. However, one can use table constructors to simulate this mechanism. As an example, suppose a function to concatenate all its arguments. It could be written like

```
function concat (o)
  local i = 1
  local s = ''
  while o[i] do
    s = s .. o[i]
    i = i+1
  end
  return s
end
```

To call it, one uses a table constructor to join all arguments:

```
  x = concat{"hello ", "john", " and ", "mary"}
```

8.5 Persistence

Because of its reflexive facilities, persistence in Lua can be achieved within the language. 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(format('\n%s =', 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, 'q')
  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 that all indices are identifiers (that is, the tables are being used as records), then 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, 'q')
  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
```

8.6 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 illustrated in Figure 1. This code must be registered with:

```
lua_pushstring("parent");
lockedParentName = lua_ref(1);
lua_pushobject(lua_setfallback("index", Index));
lockedOldIndex = lua_ref(1);
```

Notice how the string `"parent"` is kept locked in Lua for optimal performance.

8.7 Programming with Classes

There are many different ways to do object-oriented programming in Lua. This section presents one possible way to implement classes, using the inheritance mechanism presented above. *Please notice: the following examples only work with the index fallback redefined according to Section 8.6.*

As one could expect, a good way to represent a class is with a table. This table will contain all instance methods of the class, plus optional default values for instance variables. An instance of a class has its `parent` field pointing to the class, and so it "inherits" all methods.

For instance, a class `Point` can be described as in Figure 2. Function `create` helps the creation of new points, adding the parent field. Function `move` is an example of an instance method. Finally, a subclass can be created as a new table, with the `parent` field pointing to its superclass. It is interesting to notice how the use of `self` in method `create` allows this method to work properly even when inherited by a subclass. As usual, a subclass may overwrite any inherited method with its own version.

8.8 Modules

Here we explain one possible way to simulate modules in Lua. The main idea is to use a table to store the module functions.

A module should be written as a separate chunk, starting with:

```
#include "lua.h"

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

void callOldFallback (lua_Object table, lua_Object index)
{
  lua_Object oldIndex = lua_getref(lockedOldIndex);
  lua_pushobject(table);
  lua_pushobject(index);
  lua_callfunction(oldIndex);
  if (lua_getresult(1) != LUA_NOOBJECT)
    lua_pushobject(lua_getresult(1));  /* return result */
}

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_pushref(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.

```
Point = {x = 0, y = 0}

function Point:create (o)
  o.parent = self
  return o
end

function Point:move (p)
  self.x = self.x + p.x
  self.y = self.y + p.y
end

...

--
-- creating points
--
p1 = Point:create{x = 10, y = 20}
p2 = Point:create{x = 10}  -- y will be inherited until it is set

--
-- example of a method invocation
--
p1:move(p2)
```

Figure 2: A Class `Point`.

```
function open (mod)
  local n, f = next(mod, nil)
  while n do
    setglobal(n, f)
    n, f = next(mod, n)
  end
end
```

Figure 3: Opening a module.

```
if modulename then return end  -- avoid loading twice the same module
modulename = {}  -- create a table to represent the module
```

After that, functions can be directly defined with the syntax

```
function modulename.foo (...)
  ...
end
```

Any code that needs this module has only to execute `dofile("filename")`, where `filename` is the file where the module is written. After this, any function can be called with

```
modulename.foo(...)
```

If a module function is going to be used many times, the program can give a local name to it. Because functions are values, it is enough to write

```
localname = modulename.foo
```

Finally, a module may be *opened*, giving direct access to all its functions, as shown in the code in Figure 3.

8.9 A CFunction

A CFunction to compute the maximum of a variable number of arguments is shown in Figure 4. 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
```

8.10 Calling Lua Functions

This example illustrates how a C function can call the Lua function `remove_blanks` presented in Section 8.3.

```
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 4: C function `math_max`.

```
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' */
}
```

9 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.

`-` runs interactively, accepting commands from standard input until an `EOF`.

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

`var=exp` executes `var=exp` as a Lua chunk.

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

All arguments are handle in order. For instance, an invocation like

```
$ lua - a=1 prog.lua
```

will first interact with the user until an `EOF`, then will set `a` to 1, and finally will run file `prog.lua`.

Please notice that the interaction with the shell may lead to unintended results. For instance, a call like

```
$ lua a="name" prog.lua
```

will *not* set `a` to the string `"name"`. Instead, the quotes will be handled by the shell, lua will get only `a=name` to run, and `a` will finish with **nil**. Instead, one should write

```
$ lua 'a="name"' prog.lua
```

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. 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 2.4

The whole I/O facilities have been rewritten. We strongly encourage programmers to addapt their code to this new version. However, we are keeping the old version of the libraries in the distribution, to allow a smooth transition. The incompatibilities between the new and the old libraries are:

- The format facility of function `write` has been supersed by function `format`; therefore this facility has been dropped.
- Function `read` now uses *read patterns* to specify what to read; this is incompatible with the old format options.
- Function `strfind` now accepts patterns, so it may have a different behavior when the pattern includes special characteres.

Incompatibilities with version 2.2

- Functions `date` and `time` (from `iolib`) have been superseded by the new, more powerful version of function `date`.
- Function `append` (from `iolib`) now returns 1 whenever it succeeds, whether the file is new or not.
- Function `int2str` (from `strlib`) has been superseded by new function `format`, with parameter `"%c"`.
- The API lock mechanism has been superseded by the reference mechanism. However, `lua.h` provides compatibility macros, so there is no need to change programs.
- The API function `lua_pushliteral` now is just a macro to `lua_pushstring`.

Incompatibilities with version 2.1

- The function `type` now returns the string `"function"` both for C and Lua functions. Because Lua functions and C functions are compatible, this behavior is usually more useful. When needed, the second result of function `type` may be used to distinguish between Lua and C functions.
- A function definition only assigns the function value to the given variable at execution time.

Incompatibilities with version 1.1

- 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.*
- 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 corresponding call, with no intermediate calls to Lua. Special care should be taken with macros like `lua_getindexed` and `lua_getfield`.

Index

Contents