Reference Manual of the Programming Language Lua

Roberto Ierusalimschy
Luiz Henrique de Figueiredo
Waldemar Celes Filho

TeCGraf — PUC-Rio
roberto, lhf, celes@icad.puc-rio.br

May 27, 1994

Abstract

Lua is an embedded programming language designed to be used as a configuration language for any program that needs one. This document describes 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 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 embedded 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. Lua was designed by R. Ierusalimschy, L. H. de Figueiredo and W. Celes, and implemented by W. Celes.

Lua is implemented as a library, written in C. Being an embedded 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_1.0.tar.Z
```

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/or 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 in Lua is defined, its body is compiled and stored in a global variable with the given name.

[1] As usual in extended BNF, $\{a\}$ 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.

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 valid operations in Lua, besides assignment and equality test.

The type *table* implements associative arrays, that is, arrays that can be indexed both with numbers and with strings. 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"]`.

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.

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. More specifically, the string is converted to a number using the standard `strtod` C function. 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. 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 also occurs in other contexts, such as multiple assignment.

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.

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

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.

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

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).

A single name can denote a global or a local variable.

$$\begin{array}{rcl} var & \rightarrow & name \\ var & \rightarrow & var\ \text{'['}\ exp1\ \text{']'}\ \mid var\ \text{'.'}\ name \end{array}$$

Brackets are used to index a table. In this case, `var` must result in a table value; otherwise, there is an execution error. 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.

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

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:

$$
\mathit{ret} \quad \rightarrow \quad \textbf{return}\ \mathit{explist}
$$

4.4.4 Expressions as Statements

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

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

Eventual returned values are thrown away. Function calls are explained in Section 4.5.8, while 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 block end. 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:

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

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, with the usual meaning. These operators are the binary `+`, `-`, `*` and `/`, and the unary `+` and `-`. The operands must be numbers, or strings that can be converted to numbers, according to the rules given in Section 4.2.

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 can only be applied to strings and numbers. If one of the arguments is a string, the other is converted to a string, and their values are compared using lexicographical order. Otherwise, both are numbers and are compared as such.

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 "`..`". The operands must be strings or numbers, which are converted to strings according to the rules in Section 4.2.

4.5.6 Precedence

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

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

All binary operators are left associative.

4.5.7 Table Constructors

Table constructors are expressions that create tables. Table constructors are offered in different flavors. The simplest one is:

tableconstructor $\rightarrow$ '@' '(' [*exp1*] ')'

Such an expression results in a new empty table. An optional dimension may be given as a hint to the initial table size. Independently of the initial dimension, all arrays in Lua stretch dynamically as needed.

To construct a table and initialize some fields, the following syntax is available:

tableconstructor → '@' [*name*] *fieldlist*

Such an expression creates a new table, which will be its final value, initialize some of its fields according to `fieldlist` (see below), and, if `name` is given, calls a function with that name passing the table as parameter. This function can be used to check field values, to create default fields, or for any other side-effect.

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

This field list initializes named fields in a table. As an example:

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

is equivalent to:

```
temp = @(2)
temp.x = 1
temp.y = 3
f(temp)
a = temp
```

In order to initialize a list, one can use the following syntax:

fieldlist → '[' [*lfieldlist1*] ']'
lfieldlist1 → *exp* {',' *exp*}

As an example:

```
a = @["v1", "vv"]
```

is equivalent to:

```
temp = @(2)
temp[1] = "v1"
temp[2] = "v2"
a = temp
```

As particular cases, the following two expressions are completely equivalent:

```
@f{ }        @f[ ]
```

4.5.8 Function Calls

A function call has the following syntax:

functioncall → *var* '(' [*explist1*] ')'
explist1 → {*exp1* ','} *exp*

Here, `var` can be any variable (global, local, indexed, etc) whose value has type *function* or *Cfunction*. All argument expressions are evaluated before the call, from left to right; 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 parameters.

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; functions cannot be defined inside other functions. 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 the control reaches the end of a function without a return instruction, the function returns with no results.

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. error handling.

All API functions are declared in the file `lua.h`. Unless stated otherwise, API functions return an error code: 0 in case of success, non 0 in case of errors.

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);
```

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_Object` is declared as:

```
typedef struct Object *lua_Object;
```

where Object is not declared in lua.h.

Lua has garbage collection. Therefore, there is no guarantee that a lua_Object will be valid after another execution of Lua code. A good programming practice is to convert such objects to C values as soon as they are available, and never store them in global variables.

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

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

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

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

```
float          lua_getnumber         (lua_Object object);
char          *lua_getstring         (lua_Object object);
char          *lua_copystring        (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 another execution of Lua code. The function lua_copystring behaves exactly like lua_getstring, but returns a fresh copy of the string.

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, the conversion from a specific C type to the type lua_Object, is done by using the following functions:

```
int            lua_pushnumber        (float n);
int            lua_pushstring        (char *s);
int            lua_pushcfunction     (lua_CFunction f);
int            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). To complete the set, the value **nil** or a lua_Object can also be pushed onto the stack, with:

```
int            lua_pushnil           (void);
int            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:

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

Tables can also be manipulated via the API. Given a table, the functions

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

return the contents of an index. The first one is used for numeric indices, while the second can be used for any string index. As in Lua, if the index is not present in the table, then the returned `lua_Object` has value **nil**.

To store a value, previously pushed onto the stack, in a position of a table, the following functions are available:

```
int            lua_storeindexed         (lua_Object object, float index);
int            lua_storefield           (lua_Object object, char *field);
```

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

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. Then, the function is called using:

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

where the second argument (`nparam`) is the number of values pushed onto the stack. Finally, the returned values (a Lua function may return many values) are popped from the stack in reverse order, i.e., the last result is popped first. Popping is done with the function

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

When there are no more results to be popped, this function returns 0.

An example of C code calling a Lua function is shown in 7.5.

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 0. 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 7.4 presents an example of a Cfunction.

5.6 Error Handling

Whenever an error occurs during Lua compilation or execution, an error routine is called, and the corresponding `lua_dofile` or `lua_dostring` is terminated returning an error condition.

The only argument to the error routine 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). The standard error routine only prints this message in the standard error output. If needed, it is possible to set another error routine, using the function:

```
void           lua_errorfunction       (void (*fn) (char *s));
```

whose argument is the address of the new error function.

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. Such 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 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'
- 'userdata'

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)`

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**.

`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 sufix of `s`.

`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. It provides the following functions:

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

The functions `floor`, `sqrt`, `pow`, `ceil`, `sin`, `cos`, `tan`, `asin`, `acos`, and `atan` are only interfaces to the homonymous functions in the C library, with the difference that, in 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`.

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 i = 1
  while strsub(s,i,i) = ' ' do
    i = i+1
  end
  local l = strlen(s)
  while strsub(s,l,l) = ' ' do
    l = l-1
  end
  return strsub(s,i,l)
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)
    i, v = next(t, i)
    if i then write(', ') end
  end
  write('}')  -- closes constructor
end
```

7.4 A Cfunction

A Cfunction to compute the maximum of a variable number of arguments may be written as:

```
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++)) == 0)
 { lua_error ("too few arguments to function `max'"); return; }
 /* and this argument must be a number */
 if (!lua_isnumber(o))
 { lua_error ("incorrect arguments to function `max'"); return; }
 dmax = lua_getnumber (o);
 /* loops until there is no more arguments */
 while ((o = lua_getparam(i++)) != 0)
 {
  if (!lua_isnumber(o))
  { lua_error ("incorrect arguments to function `max'"); return; }
  d = lua_getnumber (o);
  if (d > dmax) dmax = d;
 }
 /* push the result to be returned */
 lua_pushnumber (dmax);
}
```

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
```

7.5 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", 1);  /* call Lua function with 1 parameter */
  strcpy(s, lua_getstring(lua_pop()));  /* copy result back to 's' */
}
```

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.