Quick Reference Guide

Programming

Lua 4

Version 4.0.1

Based on the Lua 4 Manual by
Waldemar Celes, Roberto Ierusalimschy
and Luiz Henrique de Figueiredo, TeCGraf
Adapted by Kein-Hong Man
Revision date: 2003-05-18

Contents

Conventions

`fixed`	Denotes C code, Lua code or text you enter literally.
THIS	Denotes arguments, variable text, i.e. things you must fill in.
word	Denotes functions or keywords, i.e. words with special meaning.
[...]	Denotes an optional part.
{...}	Denotes an optional and repeatable part.

1. Introduction

Lua is an extension programming language designed to support general procedural programming with data description facilities. Lua is intended to be embedded as a powerful, light-weight configuration language.

Lua is implemented as a library, written in C. Lua has no notion of a "main" program: it only works *embedded* in a host client, called the *embedding* program. This host program controls Lua via an API. Lua can be augmented to cope with a wide range of different domains, creating customized languages sharing a syntactical framework.

Lua is free-distribution software, and is provided as usual with no guarantees, as stated in its copyright notice. The canonical sources are located at:

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

Basic contact information:

`lua@tecgraf.puc-rio.br`	(comments, questions, and bug reports)
`lua-l@tecgraf.puc-rio.br`	(mailing list)
`http://www.tecgraf.puc-rio.br/lua/lua-l.html`	(ditto)

The Lua language and its implementation have been entirely designed and written by Waldemar Celes, Roberto Ierusalimschy and Luiz Henrique de Figueiredo at TeCGraf, the Computer Graphics Technology Group, Department of Computer Science, of PUC-Rio (the Pontifical Catholic University of Rio de Janeiro) in Brazil.

2. Environments and Chunks

All statements in Lua are executed in a *global environment*, initialized via a call to **`lua_open`**. The environment persists until a call to **`lua_close`**, or the end of the embedding program. Multiple independent global environments are supported.

The global environment can be manipulated by the embedding program via *API functions* from the library that implements Lua.

Any variable is assumed to be *global* unless explicitly declared *local*. Before the first assignment, the value of a global variable is **nil**. A table is used to keep all global names and values.

The unit of execution of Lua is called a *chunk*. A chunk is simply a sequence of statements, which are executed sequentially. Each statement can be optionally followed by a semicolon.

A chunk may be stored in a file or in a string. All modifications a chunk effects on the global environment persist after the chunk ends. Lua precompiles chunks into bytecodes for the Lua virtual machine prior to execution. Precompiled binary chunks (using `luac`) can be stored in files and used interchangeably with text chunks; detection is automatic.

3. Types and Tags

Types in Lua

Lua is *dynamically typed*. All values carry their own type and a *tag*. The six basic types are: *nil*, *number*, *string*, *function*, *userdata*, and *table*. The **type** function returns a string describing the type of a given value.

nil	Type of **nil**, which is different from any other value.
number	Double-precision (64 bit) floating-point numbers.
string	Character strings. Strings may contain any 8-bit character, including embedded nulls. When used for text, strings are platform- and locale-dependent.
function	Functions are *first-class values* in Lua. They can be stored in variables, passed as arguments, and returned as results. Functions have two different tags. All Lua functions have the same tag, and all C functions have the same tag.
userdata	This type is provided to allow arbitrary C pointers to be stored in Lua variables; corresponds to a `void*` and has no pre-defined operations in Lua, except assignment and equality.
table	This type implements associative arrays. Arrays can be indexed with any value (except **nil**). Use *table* to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc. This is the main data structuring mechanism in Lua.

More about Tables

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

To represent records, the field name is used as an index. Tables may also carry *methods*, allowing object-oriented programming. The following are equivalents (syntactic sugar) for forming records and methods:

```
a.name  <->  a["name"]
t:f(x)  <->  t.f(t,x)
```

Tag Methods

Each of the types *nil*, *number*, and *string* has a different tag. All values of each of these types have the same pre-defined tag. Values of type *function* can have two different tags, depending on whether they are Lua functions or C functions. Values of type *userdata* and *table* can have variable tags, assigned by the programmer.

The **tag** function returns the tag of a given value. User tags are created with the function **newtag**. The **settag** function is used to change the tag of a table. The tag of userdata values can only be set from C. Tags are mainly used to select *tag methods* when some events occur. Tag methods are the main mechanism for extending the semantics of Lua.

4. Lexical Conventions

Reserved Words and Other Tokens

Identifiers in Lua can be any string of letters, digits, and underscores, not beginning with a digit. Any character considered alphabetic by the current locale can be used in an identifier. The following words are *reserved*, and cannot be used as identifiers:

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

Lua is *case-sensitive*. `and`, `And` and `ánd` are different, valid identifiers. By convention, names of internal variables start with an underscore followed by uppercase letters (e.g. `_INPUT`). The following strings denote other tokens:

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

Comments

Comments start with a double hyphen (`--`) and run until the end of the line. The first line of a chunk is skipped if it starts with # (for Unix scripting.)

Coercion and Adjustment

A string is converted to a number if it is used in an arithmetic operation. A number is converted to a string if it is used as a string. The *exact* value of the number is preserved. (Use **format** for printing numbers instead.)

A list of values used in multiple assignments or in function calls is *adjusted* at run time in order to match requirements. Excess values are *thrown away*. If there is a shortage, the list is extended with as many **nil**s as needed.

5. Literals

Literal strings can be delimited by matching single or double quotes, and can contain the following C-like escape sequences:

`\a`	bell	**`\v`**	vertical tab
`\b`	backspace	**`\\`**	backslash
`\f`	form feed	**`\"`**	double quote
`\n`	newline	**`\'`**	single quote
`\r`	carriage return	**`\`*`newline`***	embedded newline
`\t`	horizontal tab	**`\`*`ddd`***	*ddd* is decimal value of character

Literal strings can also be delimited by matching `[[ … ]]`. This form may run for several lines, may contain nested `[[ … ]]` pairs, and do not interpret escape sequences. Convenient for writing program code or other quoted strings. *Numerical constants* may have an optional decimal part and an optional decimal exponent.

6. Statements

Blocks

A block is a list of statements; syntactically, a block is equal to a chunk. A block may also be explicitly delimited:

do BLOCK **end**

Explicit blocks are useful to control the scope of local variables, or to add a **return** or **break** statement in the middle of another block.

Assignment

Lua allows multiple assignment. The syntax for assignment defines a list of variables on the left side and a list of expressions on the right side:

```
VARLIST = EXPLIST
VAR { , VAR } = EXP { , EXP }
```

Lua first evaluates all values on the right side and eventual indices on the left side, and then makes the assignments. Thus, multiple assignment also can be used to *exchange* two or more values:

```
VAR1, VAR2 = VAR2 , VAR1
```

If the two lists have different lengths, an *adjustment* is made before assignment.

Denoting Variables

A single name can denote a global variable, a local variable, or a formal parameter. Square brackets are used to index a table:

```
VAR [ EXP ]
```

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

```
VAR.NAME   <->   VAR [ "NAME" ]
```

The meaning of assignments and evaluations of global variables and indexed variables can be changed by tag methods.

Local Declarations

Local variables may be declared anywhere inside a block. The declaration may include an initial assignment (which may be a multiple assignment.) Otherwise, all variables are initialized with **nil**.

```
local NAME {, NAME} [= EXPLIST]
```

A chunk is also a block, and so local variables can be declared outside any explicit block. The scope of local variables begins *after* the declaration and lasts until the end of the block.

7. Control Structures

Lua has a full complement of looping and conditional structures, similar to Pascal and C. The condition expression EXP1 of a control structure may return any *single* value. All values different from **nil** are considered true; only **nil** is considered false.

while EXP1 **do** BLOCK **end**
repeat BLOCK **until** EXP
if EXP1 **then** BLOCK
  { **elseif** EXP1 **then** BLOCK }
  [**else** BLOCK] **end**

Exiting Loops

return [EXPLIST]
break

The **return** statement is used to return values from a function or from a chunk. The **break** statement can be used to terminate the execution of a loop, skipping to the next statement after the loop. A **break** ends the innermost enclosing loop (**while**, **repeat**, or **for**).

return and **break** statements can only be written as the *last* statements of a block, otherwise an explicit inner block can used, as in the idiom '`do return end`', because now **return** is last statement in the inner block.

For Loops

for VAR **=** START**,** LIMIT [**,** STEP] **do** BLOCK **end**
for INDEX **,** VALUE **in** EXP1 **do** BLOCK **end**

Numerical Form

The behavior is *undefined* if you assign to VAR inside the block. The default step is +1. If expressions are passed, they are evaluated only once, before the loop starts. The loop variable VAR is local to the statement; you cannot use it after the loop ends. If you need the value of the index, assign it to another variable before breaking. A **break** exits a **for** loop.

Table Form

The table **for** statement traverses all pairs (INDEX, VALUE) of a given table. The behavior is *undefined* if you assign to INDEX inside the block, or if you change the table during the traversal. The variables INDEX and VALUE are local to the statement; you cannot use their values after the loop ends. If you need to use any value outside the loop, assign them to other variables before breaking. A **break** exits a **for** loop. Traversal order is undefined, *even for numerical indices* (to traverse indices in numerical order, use a numerical **for**.)

8. Expressions and Operators

The basic expressions in Lua consist of: expressions in parentheses, **nil**, numbers, literals, variables, upvalues, functions, function calls, and table constructors. The values returned by an expression must be adjusted to a single value. For relational and logical operators, **nil** is false, non-**nil** is true.

The Lua operator list and precedence, from the lower to the higher priority (use parentheses to override precedence):

Assoc	Operators	Description
left	**and or**	Logical AND, logical OR
left	**< > <= >= ~= ==**	Relational operators
left	**..**	Concatenation
left	**+ -**	Arithmetic addition, subtraction
left	*****	Arithmetic multiplication, division
right	**not - (unary)**	Logical NOT, unary minus
right	**^**	Exponentiation

Pre-compiler optimizations may change some results if you define non-associative tag methods for these operators. Optimizations do not change normal results.

An exponentiation always calls a tag method; this can be redefined.

Relational Operators

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

Coercion rules (conversion between strings and numbers) *do not* apply to equality comparisons.

The exact behaviour of the order operators (< > <= >=) can be found in the description for the **lt** tag method.

Logical Operators

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

Two useful Lua idioms that use logical operators are (where b should not be **nil**):

```
x = x or v         <->  if x == nil then x = v end
x = a and b or c   <->  if a then x = b else x = c end
```

9. Table Constructors

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

VAR **= {** FIELDLIST **}**

Initializing Tables

There are two basic ways in which table elements can be initialized. The first assigns expressions in the list to consecutive numerical indices, starting with an index value of 1:

VAR **= {** EXP { , EXP} [,] **}**
E.g. `x = {2, 3, 5, 7,}`

The second form lists key-value pairs:

VAR **= {** NAME **=** EXP {, NAME **=** EXP} [,] **}**

where NAME **=** EXP is syntactic sugar for **[**EXP**] =** EXP

E.g. `a = {[f(k)] = g(y), x = 1, y = 3, [0] = b+c}`
E.g. `x = 1` is syntactic sugar for `["x"] = 1`

In both forms, the final trailing comma is always optional. In addition, one method of construction may follow the other, separated by a semicolon. For example, all forms below are correct:

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

The field “n” in a table is used in many functions as a “size” indicator, otherwise “size” is the largest numerical index with a non-**nil** value. Where possible, use functions like **getn**, **foreachi**, **tinsert** and **tremove** instead of modifying field “n” directly.

Table Syntax

The actual syntax of table construction is given below for easy reference:

```
fieldlist ::=  lfieldlist
             | ffieldlist
             | lfieldlist ’;’ ffieldlist
             | ffieldlist ’;’ lfieldlist

lfieldlist  ::= [lfieldlist1]
ffieldlist  ::= [ffieldlist1]

lfieldlist1 ::= exp {’,’ exp} [’,’]
ffieldlist1 ::= ffield {’,’ ffield} [’,’]

ffield ::=  ’[’ exp ’]’ ’=’ exp | ’name’ ’=’ exp
```

10. Functions in Lua

Function Calls

A function call in Lua has the following syntax and alternative forms:

***varorfunc* (** {EXP1, } EXP **)**

`v:name(...)`	`v.name(v,...)`	Call a method (`v` is evaluated once)
`f{...}`	`f({...})`	Call `f` with a single new table
`f'...'`	`f('...')`	Call `f` with a single literal string
`f"..."`	`f('...')`	Call `f` with a single literal string
`f[[...]]`	`f('...')`	Call `f` with a single literal string

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

All argument expressions are evaluated *before* the call.

A function can return any number of results. The number of results must be adjusted before they are used. If the function is called as a statement, all returned values are discarded (list adjusted to 0.)

If a function is called as an argument in a place that needs a single value (EXP1) then its return list is adjusted to 1. If the function is called in a place that can hold many values (EXP), then no adjustment is made. The only places that can hold many values is the *last* (or the only) expression in an assignment, in an argument list, or in the **return** statement.

Function Definitions

A function definition is an executable expression, whose value has type *function*. Function definitions can be nested, or used in assignment statements.

function `funcname` **(** ... **)** BLOCK **end**
function `funcname` **(** NAME {, NAME} [, **...**] **)** BLOCK **end**

Syntactic sugar for function definitions and their equivalents:

function `f` **(**ARGS**)** …	`f` **=** **function** **(**ARGS**)** …
function `v.f` **(**ARGS**)** …	`v.f` **=** **function** **(**ARGS**)** …
function `v:f` **(**ARGS**)** …	`v.f` **=** **function** **(**`self,` ARGS**)** …

Whenever Lua executes the function definition, its upvalues are fixed, and the function is *instantiated* (or *closed*). This instance (or *closure*) is the final value of the expression. Different instances of the same function may have different upvalues.

An adjustment is made to the argument list if required. Parameters act as local variables. Results are returned using the **return** statement, otherwise the function returns with no results.

If the function is a *vararg function* (denoted by the '…' at the end of its parameter list,) it collects all extra arguments into an implicit table parameter, called `arg`, with a field `n` whose value is the number of extra arguments (found at positions 1, 2, …, `n`.)

11. Visibility and Upvalues

Visibility

A local variable (including parameters) in a function body has precedence over a global variable of the same name.

Global variables may be used in a function body as long as they are not *shadowed* by local variables with the same name from enclosing functions.

A function *cannot* access a local variable from an enclosing function. However, a function may access the *value* of a local variable from an enclosing function, using *upvalues.*

Upvalues

An *upvalue* of a local variable from an enclosing function is created by adding a **%** prefix to the variable name:

```
%localvarname
```

An upvalue is a variable whose value is *frozen* when the function wherein the upvalue appears is instantiated. The name used in an upvalue may be the name of any variable visible at the point where the function is defined, that is, global variables and local variables from the *immediately enclosing* function.

When the upvalue is a table, only the *reference* to that table (which is the value of the upvalue) is frozen; the table contents can be changed at will. This is a technique for having writable but private state attached to functions.

12. Error Handling

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

Memory allocation errors are an exception to the previous rule. When memory allocation fails, Lua may not be able to execute the **_ERRORMESSAGE** function. So, for memory allocation errors, the corresponding function from the library returns immediately with a special error code (`LUA_ERRMEM`).

_ERRORMESSAGE (ERRORSTRING)

> A variable holding the error handler function for non-memory errors. Can be set by the user to provide customized error handling. Accepts a string argument describing the error. By default, this function variable is set to the basic library function **_ALERT**, which prints the message to **stderr**. The standard I/O library redefines this function and uses the debug facilities to print some extra information, such as a call stack traceback.

Explicit errors can be generated by calling **error**. Lua code can “catch” an error using the function **call**. See the basic function library for a description of these functions.

13. Tag Methods

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

Behaviour

The tag method called for an event is selected according to the tag of the values involved in the event. The return value of the tag method that is called becomes the result of the event.

The handling of tag methods degrades gracefully, either by providing some default behaviour for events dealing with normal data types, or by calling the error function if no valid tag method could be selected for the event.

The following describes the selection method for tag methods for events with two operands (binary operations): First, Lua tries the first operand. If its tag does not define a tag method for the operation, then Lua tries the second operand. If it also fails, then it gets a tag method from tag 0.

Manipulation

settagmethod changes the tag method associated with a given pair *(tag, event)* and returns the previous tag method for that pair. **gettagmethod** receives a tag and an event name and returns the current method associated with the pair.

Events

Tag methods are called in the following events:

add

Called when a `+` operation is applied to non-numerical operands.
Coercion to numbers is attempted on both operands.
If both operands are valid numbers, the primitive add operation is used;
otherwise the tag method for the binary operation "add" is retrieved.
If the tag method exists, it is called with the original operands;
otherwise the error function is called with an error message.
Tag method: `tm(op1, op2, "add")`

sub

Called for a `-` operation. Behavior similar to the "add" event.

mul

Called for a `*` operation. Behavior similar to the "add" event.

div

Called for a `/` operation. Behavior similar to the "add" event.

pow

Called when a `^` operation is applied, even for numerical operands.
The tag method for the binary operation "pow" is retrieved.
If the tag method exists, it is called with the given operands;
otherwise the error function is called with an error message.
Tag method: `tm(op1, op2, "pow")`

List of events for tag methods, continued:

unm

Called when a unary `-` operation is applied to a non-numerical operand.

Coercion to number is attempted on the operand.
If the operand is a valid number, the primitive negation operation is used;
otherwise the tag method for the unary operation "unm" is retrieved.
If the tag method does not exist, get the global tag method (tag 0).
If the tag method exists, it is called with the original operand and a nil;
otherwise the error function is called with an error message.

Tag method: `tm(op1, nil, "unm")`

lt

Called when an order operation is applied to non-numerical or non-string operands. Corresponds to the < operator.

If both operands are numeric, use the numeric comparison;
else if both operands are strings, use the lexicographic comparison;
otherwise the tag method for the binary operation "lt" is retrieved.
If the tag method exists, it is called with the given operands;
otherwise the error function is called with an error message.

Tag method: `tm(op1, op2, "lt")`

The other order operators use this tag method according to the usual equivalences: `a>b <=> b<a`, `a<=b <=> not(b<a)` and `a>=b <=> not(a<b)`

concat

Called when a concatenation is applied to non-string operands.

If operands are either numbers or strings, use string concatenation;
otherwise the tag method for the binary operation "concat" is retrieved.
If the tag method exists, it is called with the given operands;
otherwise the error function is called with an error message.

Tag method: `tm(op1, op2, "concat")`

getglobal

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

Access the value of the variable in the global table.
Get the tag method using the value's tag and the event "getglobal".
If the tag method exists, it is called with the variable name and value;
otherwise the value of the variable is returned.

Tag method: `tm(varname, value)`

setglobal

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

Access the old value of the variable in the global table.
Get the tag method using the value's tag and the event "setglobal".
If the tag method exists, call it with variable name, old value, new value;
otherwise the value of the variable is set using the default function.

Tag method: `tm(varname, oldvalue, newvalue)`

List of events for tag methods, continued:

index

Called when Lua tries to retrieve the value of an index not present in a table. See the "gettable" event (below) for its semantics.

gettable

Called whenever Lua accesses an indexed variable. This method *cannot* be set for tables with the default tag.

```
Get the tag method using the table's tag and the event "gettable".
If the tag method exists, it is called with the table name and the index;
   else if the table is not of type "table", call the error function;
   otherwise get the tag method using the table's tag and event "index".
If the tag method exists, it is called with the table name and the index;
   otherwise the default function is used.
```

Tag method: `tm(table, index)`

settable

Called when Lua assigns to an indexed variable. This method *cannot* be set for tables with the default tag.

```
Get the tag method using the table's tag and the event "settable".
If the tag method exists, it is called with the table name, index, value;
   else if the table is not of type "table", call the error function;
   otherwise the default function is used.
```

Tag method: `tm(table, index, value)`

function

Called when Lua tries to call a non-function value.

```
If the function is of type "function", call the default handler function;
   else get tag method using the function's tag and the event "function".
If the tag method exists, call it indirectly with the function name inserted
into the first element of the argument table;
   otherwise the error function is called with an error message.
```

Indirect tag method: `call(tm, arg)`

gc

Called when Lua is garbage collecting a userdata. Can be set only from C, and *cannot* be set for a userdata with the default tag. For each userdata to be collected, Lua does the following:

```
Get the tag method using the object's tag and the event "gc".
If the tag method exists, it is called with object.
```

Tag method: `tm(obj)`

In a garbage-collection cycle, the tag methods for userdata are called in *reverse* order of tag creation. At the end of the cycle, Lua does the equivalent of the call `gc_event(nil)`.

14. The Lua API

The API is a set of C functions available to the host to communicate with Lua. All API functions, related types and constants are declared in `lua.h`.

Lua States

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

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

> If `stacksize` is zero, then a default size of 1024 is used. Each function call needs one stack position for each argument, local variable, and temporary value, plus one position for book-keeping. The stack must also have some 20 extra positions available.

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

> Destroys all objects in the given Lua environment (calls garbage-collection tag methods, if any) and frees all dynamic memory used by that state. Usage is not compulsory, because all resources are released when your program ends. Useful for long-running applications such as daemons or servers.

15. Manipulating The Stack

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

Query operations in the API can refer to any element in the stack by using an *index*: A positive index represents an *absolute* stack position (starting at 1); a negative index represents an *offset* from the top of the stack.

For a stack of *n* elements, index 1 represents the first element (bottom of stack), while index *n* represents the last element (top of stack); index *-1* also represents the last element, and index *-n* represents the first element.

A *valid* index must satisfy the following: `1 <= abs(index) <= top`

Any indices inside the available stack space are called *acceptable indices*. An *acceptable index* (which must be *non-zero*) can be defined as:

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

When you use the API, *you are responsible for controlling stack overflow*.

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

> Returns the number of elements in the stack (0 means an empty stack.)

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

> Returns the number of stack positions still available. Whenever Lua calls C, it ensures that at least `LUA_MINSTACK` (defined in `lua.h`, and must be >= 16) positions are still available.

Stack Manipulation

The API offers the following functions for basic stack manipulation:

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

Accepts any acceptable index, or 0, and sets the stack top to that index. If the new top is larger than the old one, then the new elements are filled with **nil**. If `index` is 0, then all stack elements are removed.

```
#define lua_pop(L,n) lua_settop(L, -(n)-1)
```

Pops `n` elements from the stack.

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

Pushes onto the stack a *copy* of the element at the given index.

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

Removes the element at the given position, shifting down the elements on top of that position to fill in the gap.

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

Moves the top element into the given position, shifting up the elements on top of that position to open space.

Querying the Stack

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

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

Returns one of the following constants, according to the type of the given object: `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`. If the index is non-valid, then the function returns `LUA_TNONE`.

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

Convert a type constant `LUA_T*` to a string. `t` is a type returned by `lua_type`. The strings returned are `nil`, `number`, `string`, `table`, `function`, `userdata`, and `no value`.

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

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

```
int lua_isnil       (lua_State *L, int index);
int lua_isnumber    (lua_State *L, int index);
int lua_isstring    (lua_State *L, int index);
int lua_istable     (lua_State *L, int index);
int lua_isfunction  (lua_State *L, int index);
int lua_iscfunction (lua_State *L, int index);
int lua_isuserdata  (lua_State *L, int index);
```

These functions return 1 if the object is compatible with the given type, and 0 otherwise. They always return 0 for a non-valid index.

Due to coercion, **`lua_isnumber`** accepts numbers and numerical strings, while **`lua_isstring`** accepts strings and numbers. **`lua_isfunction`** accepts both Lua functions and C functions. To distinguish between Lua functions and C functions, use instead **`lua_iscfunction`**. To distinguish between numbers and numerical strings, you can use **`lua_type`**.

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

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

Equivalent to **==** and **lt** in Lua, respectively. Returns 0 if any of the indices are non-valid.

Getting Values from the Stack

To translate a value in the stack to a specific C type, use:

```
double lua_tonumber (lua_State *L, int index);
```

Converts the value at the given index to a floating-point number. Must be a number or a string convertible to a number; otherwise returns 0.

```
const char *lua_tostring (lua_State *L, int index);
```

Converts the value to a C string (`const char*`). Must be a string or a number; otherwise, returns `NULL`. Returns a pointer to a string inside the Lua environment. Always null-terminated, but may also contain other zeros in their body. There is no guarantee that the pointer returned will be valid after the value is removed from the stack.

```
size_t lua_strlen (lua_State *L, int index);
```

Get a string's actual length. Helpful when strings have embedded nulls.

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
```

Converts the value indexed in the stack to a C function. Returns `NULL` if invalid type.

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

Converts value to `void*`. Returns `NULL` if invalid type.

These functions can be called with any acceptable index. When called with a non-valid index, they act as if the given value had an incorrect type.

Pushing values onto the Stack

The following functions push C values onto the stack. C values are converted to a corresponding Lua value, and the result is pushed onto the stack.

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

`lua_pushlstring` and **`lua_pushstring`** make an *internal copy* of the given string. **`lua_pushstring`** can only be used to push proper null-terminated C strings (without any embedded zeros); otherwise you should use the more general **`lua_pushlstring`**, which accepts an explicit size.

```
void lua_pushusertag (lua_State *L, void *u, int tag);
```

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

16. Miscellaneous API Functions

Garbage Collection

Two numbers control garbage collection: a byte counter of the amount of dynamic memory used, and a threshold. (The byte counter is not completely accurate.) When the threshold is crossed, Lua runs a garbage collection cycle. The byte counter is corrected, and then the threshold is reset to twice the value of the byte counter.

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

Returns the requested byte counter value in Kbytes.

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

Returns the requested threshold value in Kbytes.

`void` **`lua_setgcthreshold`** `(lua_State *L, int newthreshold);`

Sets the new threshold value in Kbytes. If the new threshold is smaller than the byte counter, then Lua immediately runs the garbage collector; after the collection, a new threshold is set according to the default rule.

To change its adaptative behavior, use the garbage-collection tag method for **nil** to set your own threshold (the tag method is called after Lua resets the threshold.)

Userdata, Tags and Tag Methods

Userdata can have different tags, whose semantics are only known to the host program. (See also **`lua_pushusertag`**)

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

Creates a new tag.

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

Changes the tag of the object (must be a userdata or a table) on top of the stack (no pop). `tag` must be a value created with **`lua_newtag`**.

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

Change tag methods. The new method is popped from the stack.

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

Gets the current value of a tag method, given the tag and the event.

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

Copy all tag methods from one tag to another. Returns `tagto`.

Executing Lua Code

A host program can execute Lua chunks written in a file or in a string. A chunk may return any number of values by pushing them onto the stack. It is up to the caller to discard results or adjust the stack space to preserve integrity.

```
int lua_dofile   (lua_State *L, const char *filename);
int lua_dostring (lua_State *L, const char *string);
```

Executes a Lua chunk. When called with `NULL`, **`lua_dofile`** executes the `stdin` stream. **`lua_dofile`** is able to execute pre-compiled chunks in addition to source code. Detection is automatic. **`lua_dostring`** executes only source code, given in textual form.

These functions return 0 in case of success, or one of the following error codes if they fail (defined in `lua.h`):

`LUA_ERRRUN`	Error while running the chunk.
`LUA_ERRSYNTAX`	Syntax error during pre-compilation.
`LUA_ERRMEM`	Memory allocation error. `_ERRORMESSAGE` handler is not called.
`LUA_ERRERR`	Error while running `_ERRORMESSAGE`.
`LUA_ERRFILE`	Error opening file (**`lua_dofile`** only.) `errno` can be checked. Call **`strerror`** or **`perror`** to inform the user.

```
int lua_dobuffer (lua_State *L, const char *buff,
                  size_t size, const char *name);
```
Can execute pre-compiled chunks like **`lua_dofile`**. Returns 0 if success, otherwise returns an error code. `name` is the "name of the chunk", used in errors and debugging. If `NULL`, a default is given.

17. Manipulating Globals and Tables

Globals can be read individually or by first getting the table of globals:

`void` **`lua_getglobal`** `(lua_State *L, const char *varname);`
Pushes onto the stack the value of the given variable. A tag method for the "getglobal" event may be triggered.

`void` **`lua_setglobal`** `(lua_State *L, const char *varname);`
Pops from the stack the value to be stored in the given variable. A tag method for the "setglobal" event may be triggered.

`void` **`lua_getglobals`** `(lua_State *L);`
Pushes the current table of globals onto the stack.

`void` **`lua_setglobals`** `(lua_State *L);`
Sets another table (popped from the stack) as the table of globals.

Lua tables can also be manipulated through the API. To read a value in a table, first the table must reside somewhere in the stack. Next, call a table API function where `index` refers to the table.

`void` **`lua_gettable`** `(lua_State *L, int index);`
Pops a key from the stack, and returns (on the stack) the contents of the table at that key. May trigger a tag method for "gettable".

`void` **`lua_rawget`** `(lua_State *L, int index);`
Gets a table value without triggering a tag method. The *raw* version.

`void` **`lua_settable`** `(lua_State *L, int index);`
First push the key and the value (in that order) to be set. The function then pops the values and sets the table. May trigger a tag method for "settable".

`void` **`lua_rawset`** `(lua_State *L, int index);`
Sets a table value without triggering a tag method. The *raw* version.

`void` **`lua_newtable`** `(lua_State *L);`
Creates a new, empty table and pushes it onto the stack.

Lua tables can be used as arrays in C, that is, tables indexed by numbers only:

`void` **`lua_rawgeti`** `(lua_State *L, int index, int n);`

Gets the value of the *n*-th element of the table at stack position `index`.

`void` **`lua_rawseti`** `(lua_State *L, int index, int n);`

Sets the value of the *n*-th element of the table at stack position `index` to the value at the top of the stack.

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

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

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

Traverse a table at stack position `index`. Pops a key from the stack, and pushes a key-value pair from the table (the “next” pair after the given key.) If there are no more elements, returns 0 and pushes nothing.

18. Manipulating Functions

Lua functions can be called using the following protocol: (a) push the function onto the stack; (b) push its arguments in *direct order*; (c) call the function using:

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

Returns the same error codes as **`lua_dostring`** and friends. `nargs` is the number of arguments that you have pushed. All arguments and the function value are popped, and the results pushed.

The number of results are adjusted to `nresults`, unless it is `LUA_MULTRET`. In that case, *all* results from the function are pushed. Results are pushed in direct order (with the last result on top.)

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

Performs the same task as above, but propagates the error, instead of returning an error code. The *raw* version.

The caller is responsible for balancing the stack. This is considered good programming practice. Some special Lua functions have their own C interfaces:

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

Generate a Lua error via C. Never returns.

If called from a C function that has been called from Lua, then the corresponding Lua execution terminates. Otherwise, the host program terminates with a call to `exit(EXIT_FAILURE)`. Before terminating, `message` is passed to the error handler function, **_ERRORMESSAGE**. If `message` is `NULL`, then **_ERRORMESSAGE** is not called.

`void` **`lua_concat`** `(lua_State *L, int n);`

Concatenates `n` values at the top of the stack, pops them, and leaves the result at the top; `n` must be at least 2. Concatenation follows the usual semantics of Lua.

Defining C Functions

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

Registers a function to Lua. Receives its Lua name, and a pointer to the function, which must have type **`lua_CFunction`**, which is defined as:

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

A C function must follow the following protocol: (a) receives its arguments from Lua in the stack, in direct order; (b) to return values to Lua, push them onto the stack, in direct order, and then return the number of results.

It is possible to associate some *upvalues* to a C function, thus creating a *C closure.* Upvalues are passed to the function whenever it is called, as ordinary arguments.

```
void lua_pushcclosure (lua_State *L, lua_CFunction fn,
                       int n);
```

Associate upvalues to a C function. First these values should be pushed onto the stack. Then call **`lua_pushcclosure`**; which pushs the C function onto the stack, with the number of associated upvalues in `n` (these are popped.) Whenever the C function is called, these upvalues are inserted as the *last* arguments to the function, after the actual arguments provided in the call. The *i*-th upvalue is in the stack at index *i-(n+1)*, where *n* is the number of upvalues.

References to Lua Objects

If your C code needs to keep a Lua value outside the life span of a C function, then it must create a *reference* to the value.

```
int lua_ref (lua_State *L, int lock);
```

Pops a value from the stack, creates a reference to it, and returns this reference. For a **nil** value, the reference is always `LUA_REFNIL`. There is also a constant `LUA_NOREF` that is different from any valid reference. If `lock` is not zero, then the object is *locked*. Locked objects will not be garbage collected. *Unlocked references may be garbage collected*.

```
int lua_getref (lua_State *L, int ref);
```

Pushes the object onto the stack. If the object has been garbage collected, returns 0 (and does not push anything.)

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

Release a reference when it is no longer needed.

Registry

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

When Lua starts, it registers a table at position `LUA_REFREGISTRY`. It can be accessed through this macro. This table can be used by C libraries as a general registry mechanism. Any C library can store data into this table, as long as it chooses a key different from other libraries.

19. The Debug Interface

Lua uses functions and *hooks* to construct different kinds of debuggers, profilers, and other tools. This interface is declared in `luadebug.h`.

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

Fills *parts* of a lua_Debug structure with an identification of the *activation record* of the function executing at a given level. Level 0 is the current running function, level *n+1* is the function that has called level *n*. Usually returns 1; if level is greater than stack depth, returns 0.

```
typedef struct lua_Debug {
  const char *event;        /* "call", "return" */
  int currentline;          /* (l) */
  const char *name;         /* (n) */
  const char *namewhat;     /* (n) global, tag method, local, field */
  int nups;                 /* (u) number of upvalues */
  int linedefined;          /* (S) */
  const char *what;         /* (S) "Lua" or "C" func, Lua "main" */
  const char *source;       /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */
  ...                       /* private part */
} lua_Debug;
```

```
int lua_getinfo (lua_State *L, const char *what,
                 lua_Debug *ar);
```

Returns 0 on error. Each character in `what` selects some fields of `ar` to be filled, as indicated by the letter in parentheses in the definition above. For example '`S`' fills in `source`, `linedefined`, and `what`. '`f`' pushes onto the stack the function that is running at the given level.

To get information about a function that is not active (that is, it is not in the stack,) you push the function onto the stack, and start the `what` string with the character `>`. The fields of `lua_Debug` have the following meaning:

`source`	If the function was defined in a string, `source` is that string; if the function was defined in a file, `source` starts with an '`@`' character followed by the file name.
`short_src`	"Printable" version of `source`, used in error messages.
`linedefined`	Line number where the definition of the function starts.
`what`	Set to "`Lua`" for a Lua function, "`C`" for a C function, or "`main`" for the main part of a chunk.
`currentline`	Current line where the given function is executing. When no line information is available, this is set to `-1`.
`name`	A reasonable name. Functions in Lua, as first class values, do not have a fixed name. If the function is a tag method, then `name` points to the event name. If the function is the value of a global variable, then `name` points to the variable name. Otherwise, `name` is set to `NULL`.
`namewhat`	Explains `name`. Set to "`global`", "`tag-method`" or "".
`nups`	Number of upvalues of a function.

These functions manipulate the local variables of a given activation record:

```
const char *lua_getlocal (lua_State *L,
                    const lua_Debug *ar, int n);
const char *lua_setlocal (lua_State *L,
                    const lua_Debug *ar, int n);
```

For local variables, indices are used. The first parameter or local variable has index 1, and so on, until the last active local variable. `ar` must be valid, filled by a previous call to **`lua_getstack`** or given as an argument to a hook. **`lua_getlocal`** gets the index of a local variable (`n`), pushes its value onto the stack, and returns its name.

For **`lua_setlocal`**, push the new value; the function assigns that value to the variable and returns its name. Both functions return `NULL` on failure; that happens if the index is out of range.

Debugging Hooks

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

Sets one of two types of debugging hooks (format defined by **`lua_Hook`**,) returning their previous values. Hooks are disabled if set to `NULL`. While Lua is running a hook, it disables other calls to hooks. Execution of hook code occurs without any further calls to hooks.

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

Line hook: Called whenever the line of code being executed is changed. The `event` field of `ar` has the string "`line`", and the `currentline` field has the line number. You can use this `ar` in other calls.

20. The Reflexive Debug Interface

The library `ldblib` provides the functionality of the debug interface to Lua programs. To use this library, open it by calling `lua_dblibopen`. This library should be used with great care; it is exclusively for debugging and similar tasks. Do *not* use them as a usual programming tool.

getinfo (FUNCTION, [WHAT])

Returns a table with information about a function. FUNCTION can also be a level number, relative to **getinfo** (level 0). If the number is invalid, a **nil** is returned. The returned table contains all the fields returned by **`lua_getinfo`**. The default for WHAT is to get all information available.

getlocal (LEVEL, LOCAL)

Returns the name and the value of the local variable with index LOCAL of the function at level level of the stack. Similar to **`lua_getlocal`**. Returns **nil** if LOCAL is out of range, and raises an error when called with a LEVEL that is out of range.

setlocal (LEVEL, LOCAL, VALUE)

Assigns VALUE to the local variable with index LOCAL of the function at level LEVEL of the stack. Returns **nil** if LOCAL is out of range, and raises an error when called with a LEVEL out of range.

setcallhook (HOOK)

Sets the function HOOK as the call hook, and returns the old hook. The only argument to the call hook is the event name (“`call`” or “`return`”.) Use **getinfo** with level 2 to get more information about the function being called or returning (level 0 is **getinfo**, level 1 is the hook.) Without arguments, call hooks are turned off.

setlinehook (HOOK)

Sets the function HOOK as the line hook, and returns the old hook. The only argument to the line hook is the line number the interpreter is about to execute. Without arguments, line hooks are turned off.

21. Lua Stand-alone

The stand-alone interpreter, `lua`, is a console-based application that can be called with any sequence of the following arguments:

`-sNUM`	Sets stack size to `NUM` (must be the first option).
`-`	Executes `stdin` as a file.
`-c`	Calls `lua_close` after running all arguments.
`-e \rm`*`stat`*	Executes string `stat`.
`-f` ***`filename`***	Executes `filename` with remaining args in table `arg`.
`-i`	Enters interactive mode with prompt.
`-q`	Enters interactive mode without prompt.
`-v`	Prints version information.
`var=`*`value`*	Sets global `var` to string `"value"`.
`filename`	Executes file `filename`.

Without arguments, the default is `"lua -v -i"` when `stdin` is a terminal, and `"lua -"` otherwise. All arguments are handled in order, except `-c`.

With `"-f` *`filename`*`"`, all remaining arguments in the command line are passed to the Lua program in a table called `arg`; the field `n` gets the index of the last argument, and the field 0 gets `"filename"`.

The **getargs** function can be used to access *all* command line arguments, with the called Lua executable file name in field `0`. Field `n` is the index of the last argument.

A multi-line statement can be written by finishing intermediate lines with a backslash (‘\’). If the global variable `_PROMPT` is defined as a string, then its value is used as the prompt, allowing the prompt to be changed.

In Unix systems, Lua scripts can be made into executable programs by using:

`chmod +x`	(sets the executable flag in Unix)
`#!/usr/local/bin/lua`	(path to Lua in the script header)
`#!/usr/local/bin/lua -f …`	(to get other arguments)

22. Standard Libraries

Lua standard libraries are provided as separate, optional C modules. To access these libraries, the corresponding initialization functions (declared in header file `lualib.h`) must be called. The current standard libraries are:

Basic library:	`void` **`lua_baselibopen`** `(lua_State *L)`
String manipulation:	`void` **`lua_strlibopen`** `(lua_State *L)`
Mathematical functions:	`void` **`lua_mathlibopen`** `(lua_State *L)`
I/O and system facilities:	`void` **`lua_iolibopen`** `(lua_State *L)`

23. Basic Function Library

_ALERT (MESSAGE)

Prints MESSAGE to `stderr`. Function may be reassigned.

assert (V [, MESSAGE])

Issues an *“assertion failed!”* error when its argument V is **nil**.

call (FUNC, ARG [, MODE [, ERRHANDLER]])

Calls FUNC with arguments in table ARG. FUNC results is returned by **call**. Error is propagated. MODE “x” protects the call; error is not propagated, **nil** is returned to signal the error. ERRHANDLER temporarily sets _ERRORMESSAGE. A **nil** disables the error handler.

collectgarbage ([LIMIT])

Sets garbage-collection threshold (Kbytes). Default is zero. A garbage collection cycle may be run if the new threshold is smaller.

copytagmethods (TAGTO, TAGFROM)

Copies all tag methods; returns TAGTO.

dofile (FILENAME)

Opens FILENAME and executes it as a Lua chunk. Auto-detects and executes pre-compiled chunks. Defaults to `stdin`. Returns **nil** if error, a non-**nil** value if the chunk returns no values, or the values returned by the chunk. Error if FILENAME is non-string.

dostring (STRING [, CHUNKNAME])

Executes STRING as a Lua chunk. Returns **nil** if error, a non-**nil** value if the chunk returns no values, or the values returned by the chunk. CHUNKNAME is used in error messages and debug information.

error (MESSAGE)

Calls error handler, then terminates the last protected function called (in C: **`lua_callfunction`**, **`lua_dofile`**, **`lua_dostring`**, **`lua_dobuffer`**, or; in Lua: **dofile**, **dostring**, or **call** in protected mode). If MESSAGE is **nil**, then **error** is not called. Never returns.

foreach (TABLE, FUNC)

Executes FUNC over all elements of TABLE. FUNC is called repeatedly with the table’s index and value pairs. Loop exits if FUNC returns any non-**nil** value, and this value is returned as the final value. Behavior is *undefined* if you change TABLE during traversal.

foreachi (TABLE, FUNC)

Executes FUNC over numerical indices of TABLE. FUNC is called repeatedly with the index and respective value as arguments, in sequential order, from `1` to `getn(table)`. Loop exits if FUNC returns any non-**nil** value, and this value is returned as the final value.

getglobal (NAME)

Gets value of a global variable, or calls a tag method for “getglobal”. NAME does not need to be a syntactically valid variable name.

getn (TABLE)

Returns the “size” of TABLE, seen as a list. If TABLE has an `n` field with a numeric value, this value is used. Otherwise, “size” is the largest numerical index with a non-**nil** value in the table.

gettagmethod (TAG, EVENT)

Returns the current tag method for a given pair *(tag, event).* Cannot be used to get a tag method for the “gc” event. (Use C API call for “gc”.)

globals ([TABLE])

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

newtag ()

Returns a new tag.

next (TABLE, [INDEX])

Traverse all fields of a table. Returns the next index of the table and its associated value. If INDEX is **nil**, returns the first index of the table and its associated value. When called with the last index, or with **nil** in an empty table, **next** returns **nil**. INDEX defaults to **nil**. **next** only considers fields with non-**nil** values. Enumeration order is not specified, *even for numeric indices.* (For numeric order, use a numerical **for** or the function **foreachi**). **next** is *undefined* if you change the table during the traversal.

print (E1, E2, …)

Prints arguments using strings returned by **tostring**. Not intended for formatted output; mainly for quickies, for instance in debugging.

rawget (TABLE, INDEX)

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

rawset (TABLE, INDEX, VALUE)

Sets the real value of `table[index]` to VALUE, without invoking any tag method. TABLE, INDEX, and VALUE must be valid.

setglobal (NAME, VALUE)

Sets NAME global variable to VALUE, or calls a tag method for “setglobal”. NAME need not be syntactically valid.

settag (T, TAG)

Sets the tag of a given table T. TAG must be a value created with **newtag**. Returns the value of its first argument (the table T.) It is impossible to change the tag of a userdata from Lua (for safety.)

settagmethod (TAG, EVENT, NEWMETHOD)

Sets a new tag method to the given pair *(tag, event)* and returns the old method. If NEWMETHOD is **nil**, then the default behavior is restored. Cannot be used for the “gc” event. (Use C API call for “gc”.)

sort (TABLE [, COMP])

Sorts TABLE in a given order, *in-place*, from `table[1]` to `table[n]`, where `n` is the result of `getn(table)`. If COMP is given, it must receive two table elements, and returns true (not **nil**) when the first is less than the second. COMP defaults to the operator **<**. The sort algorithm is *not* stable (quicksort is the current algorithm.)

tag (V)

Tests the tag of a value V; returns its tag (a number).

tonumber (E [, BASE])

Converts E to a number using optional BASE. Returns **nil** if unsuccessful. BASE can be an integer from 2 to 36 denoting [0-9A-Z]. In decimal, a fractional part and an exponent may be included. In other bases, only unsigned integers are accepted.

tostring (E)

Converts E to a string in a reasonable format. See also **format**.

tinsert (TABLE [, POS] , VALUE)

Inserts element VALUE at table position POS, shifting to open space, if necessary. Default value for POS is `getn(table)+1` (insertion at the end of the table, or append.) Table “size” (`n` field) is then updated.

tremove (TABLE [, POS])

Removes from TABLE the element at position POS, shifting to close the space, if necessary. Returns the value of the removed element. Default value for POS is the result of `getn(table)` (removal of the last element.) Table “size” (`n` field) is then updated.

type (V)

Test the type of a value; returns a string, one of: `"nil"`, `"number"`, `"string"`, `"table"`, `"function"`, and `"userdata"`.

24. String Manipulation Library

Generic functions for string manipulation. When indexing, the first character is at *position 1* (not at 0, as in C). Indices can be *negative* and are interpreted as indexing backwards, from the end of the string (e.g. the last character is at position *-1*.)

strbyte (S [, I])

Returns internal numerical code of the I-th character of S. Default of I is 1, and may be negative. *Not portable.*

strchar (I1, I2, …)

Receives 0 or more integers. Returns a string with length equal to the number of arguments, wherein each character has the internal numerical code equal to its corresponding argument. *Not portable.*

strfind (S, PATTERN [, INIT [, PLAIN]])

Looks for the first *match* of PATTERN in S. If it finds one, **strfind** returns the indices of S where this occurrence starts and ends; otherwise, it returns **nil**. If the pattern specifies captures, the captured strings are returned as extra results. INIT specifies where to start the search, defaults to 1, and may be negative. If PLAIN is 1, pattern matching facilities is turned off.

See §28 for the pattern matching format for PATTERN.

strlen (S)

Returns the length of S. An empty string has length 0. Any 8-bit character is counted, including embedded zeros.

strlower (S)

Returns a copy of S with all upper case letters changed to lower case, according to the current locale.

strrep (S, N)

Returns a string that is the concatenation of N copies of the string S.

strsub (S, I [, J])

Returns another string, which is a substring of S, starting at I and running until J; I and J may be negative. The default value of J is -1 (the end.) `strsub(s,1,j)` returns a prefix; `strsub(s, -i)` returns a suffix.

strupper (S)

Returns a copy of S with all lower case letters changed to upper case, according to the current locale.

format (FORMATSTRING, E1, E2, …)

Returns a formatted version of E1, E2, … following the description given in FORMATSTRING. Most **`printf`** rules are followed. Options/modifiers `*`, `l`, `L`, `n`, `p`, and `h` are *not supported*.

`q` formats a string suitable to be safely read back by the Lua interpreter. All double quotes, newlines and backslashes are correctly escaped. Escape sequences are recognized.

Conversions specifying the *n*-th argument uses the sequence `%d$`, where `d` is a decimal digit in the range [1-9], giving the position of the argument. E.g. For instance, `%2$d` converts the second argument.

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument, whereas `q` and `s` expect a string. The `*` modifier can be simulated by building the appropriate format string.

Neither the format string nor `%s` string values can contain *embedded zeros*. `%q` handles string values with embedded zeros.

gsub (S, PAT, REPL [, N])

Returns a *copy* of S in which all occurrences of the pattern PAT have been replaced by a replacement string specified by REPL. The second value returned is the total number of substitutions made.

See §28 for the pattern matching format for PAT.

The optional parameter N limits the maximum number of substitutions. For instance, when N is 1 only the first occurrence of PAT is replaced.

gsub (S, PAT, REPL [, N]) *(continued)*

If REPL is a string, then its value is used for replacement. Any sequence of the form %n with n between 1 and 9 stands for the value of the *n*-th captured substring, which will be substituted in.

If REPL is a function, then REPL is called *every time* a match occurs, with all captured substrings passed as arguments, in order. If the value returned by this function is a string, then it is used as the replacement string; otherwise, the replacement string is an empty string.

25. Mathematical Function Library

The mathematical function library is an interface to some functions of the standard C math library. In addition, it defines **PI** and registers a tag method for the binary operator **^** that returns *x^y* when applied to numbers x^y. The library provides the following functions:

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

The following are additional information on the variables and functions:

math_func (N)

General mathematical functions accept a single number N as the argument.

trig_func (DEG)

Trigonometrical functions accept an angle DEG expressed in *degrees*, not radians. The functions **deg** and **rad** can be used to convert between radians and degrees. (*deg = rad * 180/π*)

PI

A global variable that provides the constant value of π.

max (N, …)

Returns the maximum value of its numeric arguments.

min (N, …)

Returns the minimum value of its numeric arguments.

random ([N [, M]])

Interfaces to the ANSI C function **`rand`**. Its pseudo-random properties is implementation-dependent; determined by the compiler, operating system and platform. When called without arguments, returns a pseudo-random real number in the range *[0,1)*. When called with a number *n*, returns a pseudo-random integer in the range *[1,n]*. When called with two arguments, *n* and *m*, returns a pseudo-random integer in the range *[n,m]*.

randomseed (SEED)

Interfaces to the ANSI C function **`srand`**. Accepts one integer argument that is used to initialize the implementation-dependent random number generator.

26. Input/Output Library

Lua uses two *file handles* by default for I/O operations. These handles are stored in the global variables `_INPUT` and `_OUTPUT`. A file handle is a *userdata* containing the file stream (`FILE*`), and with a distinctive tag created by the I/O library. Unless otherwise stated, all I/O functions return **nil** on failure and some value different from **nil** on success. These facilities are broadly similar to their C function equivalents.

_INPUT

The input file handle. The default is `_INPUT=_STDIN` (**stdin**)

_OUTPUT

The output file handle. The default is `_OUTPUT=_STDOUT` (**stdout**)

_STDIN

A global variable set with the file descriptor for **stdin**.

_STDOUT

A global variable set with the file descriptor for **stdout**.

_STDERR

A global variable set with the file descriptor for **stderr**.

openfile (FILENAME, MODE)

Opens a file in the mode specified in the string MODE. Broadly equivalent to **`fopen`** in standard C. Returns a new file handle, or, in case of errors, **nil** plus a string describing the error. Does not modify either `_INPUT` or `_OUTPUT`. The MODE string can contain:

`r`	read mode	**`r+`**	update mode (all previous data preserved)
`w`	write mode	**`w+`**	update mode (all previous data erased)
`a`	append mode	**`a+`**	append update mode (previous data is preserved, append only at the end of file)
`b`	binary mode		

closefile (HANDLE)

Closes the given file. Does not modify either `_INPUT` or `_OUTPUT`.

readfrom ([FILENAME])

When called with a file name, it opens the named file, sets `_INPUT` with its handle and returns this value. Does *not* close the current input file. When called without parameters, it closes the `_INPUT` file, and restores `stdin` as the value of `_INPUT`. On error, it returns **nil**, plus a string describing the error.

If FILENAME starts with a `|` (pipe character), then input is piped via the C function **popen**. Not all systems implement pipes.

writeto (FILENAME)

With a file name, it opens the named file, sets `_OUTPUT` with its handle and returns this value. Does *not* close the current output file. An existing file will be *completely erased*. Without parameters, it closes the `_OUTPUT` file, and restores `stdout` as the value of `_OUTPUT`. On error, it returns **nil**, plus a string describing the error.

If FILENAME starts with a `|` (pipe character), then output is piped via the C function **popen**. Not all systems implement pipes.

appendto (FILENAME)

Opens FILENAME in *append* mode and sets `_OUTPUT` with its handle. On error, it returns **nil**, plus a string describing the error.

remove (FILENAME)

Deletes the file with the given name. On error, it returns **nil**, plus a string describing the error.

rename (NAME1, NAME2)

Renames file named NAME1 to NAME2. On error, it returns **nil**, plus a string describing the error.

flush ([FILEHANDLE])

Saves any written data to the given file. If FILEHANDLE is not specified, then all open files are flushed. On error, it returns **nil**, plus a string describing the error.

seek (FILEHANDLE [, WHENCE] [, OFFSET])

Sets and gets the file position (bytes), to the position given by OFFSET from a base specified by the string WHENCE, as follows:

`set`	base is position 0 (beginning of the file)
`cur`	base is current position
`end`	base is end of file

If successful, **seek** returns the final file position, measured in bytes from the beginning of the file. On error, it returns **nil**, plus a string describing the error. Default for WHENCE is `cur`; for OFFSET is 0.

`seek(file)` returns the current file position, without changing it.
`seek(file, "set")` sets the position to the beginning (returns 0.)
`seek(file, "end")` sets the position to the end (returns its size.)

tmpname ()

Returns a string with a file name that can safely be used as a temporary file. Must be explicitly opened before use; removed when not needed.

read ([FILEHANDLE,] [FORMAT1, …])

Reads file `_INPUT` (or `filehandle`, if given) according to the given formats. For each format, a string (or a number) is returned with the characters read, or **nil** if it cannot read data with the specified format.

The available formats are:

`*n`	reads a number, and returns a number
`*l`	reads the next line (skipping the end of line), or **nil** on EOF (*default behaviour*)
`*a`	reads the whole file, starting at the current position. On EOF, it returns an empty string
`*w`	reads the next word (maximal sequence of non-whitespace characters), skipping spaces if necessary, or **nil** on EOF.
`number`	reads a string with up to that number of characters, or **nil** on EOF

write ([FILEHANDLE,] VALUE1, …)

Writes the value of each argument to file `_OUTPUT` (or `filehandle`, if given.) Must be strings or numbers, otherwise use **tostring** or **format** first. On error, it returns **nil**, plus a string describing the error.

27. System Facilities

clock ()

Returns an approximation of CPU time (in sec) used by the program.

date ([FORMAT])

Returns a string containing date and time formatted according to the given FORMAT, following the rules of the ANSI C function **`strftime`**. By default, it returns a reasonable date and time representation that depends on the host system and on the current locale.

execute (COMMAND)

Equivalent to the C function **`system`**. It passes COMMAND to be executed by an operating system shell. Returns a system-dependent status code.

exit ([CODE])

Calls the C function **`exit`**, with an optional CODE, to terminate the program. Default value is the success code.

getenv (VARNAME)

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

setlocale (LOCALE [, CATEGORY])

Interfaces to the ANSI C function **`setlocale`**. LOCALE is a string specifying a locale; CATEGORY is an optional string describing which category to change: `"all"` (the default), `"collate"`, `"ctype"`, `"monetary"`, `"numeric"`, or `"time"`. Returns the name of the new locale, or **nil** if the request is invalid.

28. Pattern Matching

Patterns

A *pattern* is a sequence of *pattern items* (see below). A **^** at the beginning of a pattern anchors the match at the beginning of the subject string. A **$** at the end of a pattern anchors the match at the end of the subject string. At other positions, **^** and **$** have no special meaning and represent themselves.

Pattern Items

There are three main types of *pattern items*:

Character class patterns. A pattern item may be a single character class that matches any single character in the class. It can be optionally followed by a suffix:

*****	0 or more repetitions, matches longest possible sequence
+	1 or more repetitions, matches longest possible sequence
-	0 or more repetitions, matches shortest possible sequence
?	0 or 1 occurrence

Pattern Items, continued:

Captured patterns. A pattern item can also be in the form `%n`, for ***n*** between 1 and 9; such an item matches a sub-string equal to the *n*-th captured string.

Balanced patterns. The final form of a pattern item is `%bxy`, where ***x*** and ***y*** are two distinct characters; this matches strings that start with ***x***, end with ***y***, where the ***x*** and ***y*** are *balanced*. Pairs of string delimiters and parentheses in arithmetic expressions commonly exhibit this trait, and may be matched using such a pattern item. E.g. "`%b<>`".

Captures

If a pattern contains sub-patterns enclosed in parentheses, they describe *captures.* When a match succeeds, the sub-strings of the subject string that match captures are stored (*captured*) for future use. Captures are numbered according to their left parentheses, starting from 1. Captured strings can be used in further matches or in substitutions.

Character Classes

A *character class* is used to represent a set of characters. The following combinations are allowed in describing a character class:

`%a`	letters	`%s`	space characters
`%c`	control characters	`%u`	upper case letters
`%d`	digits	`%w`	alphanumeric characters
`%l`	lower case letters	`%x`	hexadecimal digits
`%p`	punctuation characters	`%z`	character with representation 0

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

x	Represents a literal character, where `x` is a non-magic character (`^$()%.[]*+-?`)
`.`	A dot represents all characters
`%`***x***	Represents the character `x`, where `x` is any non-alphanumeric character; used to escape the magic characters. (Any punctuation character should be preceded by a `%` when used to represent itself in a pattern.)
`[char-set]`	Represents the class which is the union of all characters in `char-set`. Ranges may be specified using a `-` (dash). `%x` classes described above may also be used as components. All other characters represent themselves. Interaction between ranges and classes is not defined.
`[^char-set]`	Represents the *complement* of `char-set`, where `char-set` is interpreted as above.

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

The definitions of letter, space, etc. depend on the current locale. In particular, the class `[a-z]` may not be equivalent to `%l`. The second form should be preferred for portability.

29. Incompatibilities with Lua 3.2

Language incompatibilities with version 3.2

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

Library incompatibilities with version 3.2:

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

API incompatibilities with version 3.2

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

30. The Complete Syntax of Lua

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

This booklet is based on the Lua 4 HTML manual from the Lua 4.0.1 distribution, timestamped 2000-11-06 17:37:03 EDT, by Luiz Henrique de Figueiredo. Feedback can be sent to Kein-Hong Man at <mkh@pl.jaring.my>.