Quick Reference Guide

Programming

Lua 5

Version 5.0.2

Based on the Lua 5 Manual by
Waldemar Celes, Roberto Ierusalimschy
& Luiz Henrique de Figueiredo, Tecgraf
Adapted by Kein-Hong Man
Revision date: 2004-05-08

Contents

Conventions

`fixed`	C code, Lua code or text you enter literally.
THIS	Arguments, variable text, i.e. things you must fill in.
word	Functions or keywords, i.e. words with special meaning.
[...]	An optional part.
{...}	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 used 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 or the *host*. 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 software, and is provided as usual with no guarantees. Lua is licensed under the terms of the MIT license. The official URL is:

```
http://www.lua.org/
```

Up-to-date information about Lua-related resources can be found at the `lua-users` wiki:

```
http://lua-users.org/
```

The Lua language and its implementation have been 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. 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 *keywords* are reserved:

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

Lua is *case-sensitive*. By convention, identifiers starting with an underscore followed by uppercase letters (such as `_VERSION`) are reserved for internal variables used by Lua. The following strings denote other tokens:

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

Literals

Literal strings can be delimited by matching single or double quotes, and can contain any 8-bit value, including embedded zeros, and the following C-like escape sequences:

\a	bell	****	backslash
\b	backspace	**\"**	quotation mark
\f	form feed	**\'**	apostrophe
\n	newline	**\[**	left square bracket
\r	carriage return	**\]**	right square bracket
\t	horizontal tab	***\newline***	embedded newline
\v	vertical tab	***\ddd***	*ddd* is decimal value of char

Literal strings can also be delimited by matching `[[ … ]]` (multiline, may be nested, does not interpret escape sequences.) When the opening '`[[`' is immediately followed by a newline, the newline is ignored. *Numerical constants* may have an optional fractional part and an optional decimal exponent.

Comments

A *short comment* starts with a double hyphen (`--`) and runs until the end of the line. A *long comment* starts with '`--[[`' and is delimited by '`]]`' (may be multiline and nested with `[[ … ]]` pairs.) The first line of a chunk is skipped if it starts with # (for Unix scripting.)

3. Values and Types

Lua is *dynamically typed*. Only values carry their own type. Lua does not have type definitions. The eight basic types are:

nil	Type of **nil**, which is different from any other value.
boolean	Type of the values **false** and **true**. Both **nil** and **false** make a condition false; any other value makes it true.
number	Double-precision floating-point numbers.
string	Arrays of characters. May contain any 8-bit character, including embedded nulls.
function	Functions are *first-class values* in Lua. Can be stored in variables, passed as arguments, and returned as results. Lua and C functions can be called and manipulated.
userdata	This type is provided to allow arbitrary C data to be stored in Lua variables. Corresponds to a block of raw memory and has no pre-defined operations except assignment and identity test.
thread	Represents independent threads of execution; for coroutines.
table	Implements associative arrays. Tables can be indexed with any value (except **nil**). Tables can be *heterogeneous*; they can contain values of all types (except **nil**). Sole data structuring mechanism in Lua; may be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc.

More about Types

The **type** function returns a string describing the type of a given value. The data type for numbers may be easily changed by recompiling Lua.

By using *metatables*, operations for userdata values can be defined. Userdata values cannot be created or modified in Lua, only through the C API. This guarantees data integrity.

To represent records, the field name is used as an index. `a.name` is provided as syntactic sugar for `a["name"]`. The value of a table field can be of any type (except **nil**). Table fields may contain functions, and carry *methods.*

Tables, functions, and userdata values are *objects*: variables contain only *references* to them. Assignment, parameter passing, and function returns always manipulate references to such values and do not imply any kind of copy.

Coercion

At run time, a string is converted to a number if it is used in an arithmetic operation, and vice versa. A reasonable format preserving the *exact* value of the number is used. (Use **format** for printing numbers instead.)

4. Variables

There are three kinds of variables in Lua: global variables, local variables, and table fields. Variables are global unless explicitly declared local. Local variables are *lexically scoped*: they can be freely accessed by functions defined inside their scope. Before the first assignment, their values are **nil**.

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 accesses to global variables and table fields can be changed via metatables. For example, an access to an indexed variable `t[i]` is equivalent to a call `gettable_event(t,i)`.

Environments

All global variables live as fields in Lua tables, called *environment tables* or simply *environments*. C functions exported to Lua all share a common *global environment*. Each Lua function has its own reference to an environment. A function inherits the environment from the function that created it. To change or get the environment table of a Lua function, call `setfenv` or `getfenv`.

The following are equivalent for global variables:

```
x
_env.x
gettable_event(_env, "x")
```

5. Statements

Chunks

The unit of execution of Lua is called a *chunk*. A chunk is simply a sequence of statements, executed sequentially. Each statement can be optionally followed by a semicolon. Lua handles a chunk as the body of an anonymous function. Chunks can define local variables and return values. A chunk may be stored in a file or in a string. Precompiled binary chunks (using `luac`) can be used interchangeably with chunks in source form; detection is automatic.

Blocks

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

do BLOCK **end**

Explicit blocks are useful to control the scope of variable declarations, 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:

VAR { **,** VAR } **=** EXP { **,** EXP }

Before the assignment, the list of values is *adjusted* to the length of the list of variables. Excess values are thrown away. If there is a shortage, the list is extended with as many **nil**s as needed. Lua first evaluates all expressions, and only then are the assignments made. Thus the following is an *exchange*:

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

The meaning of assignments to global variables and table fields can be changed via metatables. The following are equivalent:

```
t[i] = val   ↔   settable_event(t,i,val)
```

The following global variable assignments are equivalent:

```
x = val
_env.x = val
settable_event(_env, "x", val)
```

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. Such local variables die when the chunk ends.

6. Control Structures

Control structures in Lua have the usual meaning and familiar syntax:

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

Lua also has a **for** statement, see below.

The condition expression EXP of a control structure may return any value. Both **false** and **nil** are considered false; other values are considered true, including the number 0 and the empty string.

Exiting Loops

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

return [EXPLIST]
break

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

For Statement

The **for** statement has two forms, one numeric and one generic:

for VAR **=** START**,** LIMIT [**,** STEP] **do** BLOCK **end**

The default step is 1. All control expressions are evaluated only once to result in numbers, before the loop starts. The behavior is *undefined* if you assign to VAR inside the block. A **break** exits a **for** loop. VAR is local to the statement; if you need the value of the index, assign it to another variable before breaking or exiting.

for VAR1 { **,** VAR } **in** EXPLIST **do** BLOCK **end**

Works over functions, called *iterators*. For each iteration, it calls its iterator function to produce a new value, stopping when the new value is **nil**.

EXPLIST is evaluated once, giving an *iterator* function, a *state*, and an initial value for VAR1. The iterator function is called with the state and VAR1, and the results are assigned to the loop variables.

Behavior is *undefined* if you assign to VAR1 inside the block. A **break** exits a **for** loop. Loop variables are local to the statement; if you need their values, assign them to other variables before breaking or exiting the loop.

7. Expressions and Operators

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

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

An expression enclosed in parentheses always results in only one value (the first value returned or **nil** if no value.)

Relational Operators

Relational operators always result in **false** or **true**. Equality (==) first compares the tags of its operands. If types are different, the result is **false**. Otherwise, their values are compared. Numbers and strings are compared in the usual way. Objects (tables, userdata, threads, and functions) are compared by *reference.*

The operator ~= is exactly the negation of equality (==). Coercion do not apply to equality comparisons. `"0"==0` evaluates to **false**.

Order operators (< > <= >=) compare pairs of numbers; pairs of strings (using the current locale) or uses the 'lt' or the 'le' metamethod.

Logical Operators

Logical operators consider both **false** and **nil** as false and anything else as true. **not** always returns **false** or **true**.

and returns its first argument if this value is **false** or **nil**; otherwise, **and** returns its second argument. **or** returns its first argument if this value is different from **nil** and **false**; otherwise, **or** returns its second argument. Both operators use short-cut evaluation.

The following are useful Lua idioms that use logical operators (where `b` should not be **nil** or **false**):

```
x or error()      ↔  if not(x) then error() end
x = x or v        ↔  if not(x) then x = v end
x = a and b or c  ↔  if a then x = b else x = c end
```

8. 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 **= {** FIELD **{ ,** FIELD **} [,] }**
FIELD → **[** EXP **] =** EXP | NAME **=** EXP | EXP

The final trailing comma is always optional. Different forms for specifying fields can be mixed. Semicolons can be used in place of commas and mixed with commas in a table constructor.

Each field of the form **[** EXP1**] =** EXP2 adds to the table an entry with a *key* EXP1 and a *value* EXP2. The form NAME **=** EXP is equivalent to **["** NAME **"] =** EXP.

Fields of the form EXP are equivalent to **[** INDEX **] =** EXP, where INDEX are consecutive numerical integers, starting with 1. Fields in the other formats do not affect this counting.

If the last field in the list has the form EXP and the expression is a function call, then all values returned by the call enter the list consecutively. To avoid this, enclose the function call in parentheses.

Table Examples

```
x = {}
x = {2, 3, 5, 7,}
a = {[f(k)] = g(y), x = 1, y = 3, [0] = b+c}
x = {type="list"; "a", "b"}
x = {f(0), f(1), f(2),; n=3,}
```

```
a = {[f(1)] = g; "x", "y"; x = 1, f(x), [30] = 23; 45}
```
→ `45` will be placed into `a[4]`

9. Functions Calls and Definitions

Function Calls

A function call in Lua has the following syntax:

PREFIXEXP **[:** NAME **]** ARGS
ARGS → **(** EXPLIST **)** | TABLECONSTRUCTOR | LITERAL

First PREFIXEXP and ARGS are evaluated. If PREFIXEXP has type *function*, then that function is called with the given ARGS. Otherwise, its "call" metamethod is called, having as first parameter the value of PREFIXEXP, followed by 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.

If called inside another expression or in the middle of a list of expressions, then its return list is adjusted to one element (the first one). If the function is called as the last element of a list of expressions, then no adjustment is made (unless enclosed in parentheses).

The following is a summary of syntactic sugar forms:

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

A line break cannot be put before the '(' in a function call, to avoid some ambiguities. A semicolon can be added to disambiguate breaks.

Lua implements *proper tail calls* (or *proper tail recursion*). A tail call erases any debug information about the calling function, and can only happen with a particular syntax:

return FUNCTIONCALL

Function Definitions

A function definition is an executable expression, whose value has type *function.* The syntax for function definition is:

function (NAMELIST [, ...] | ... **)** BLOCK **end**

Syntactic sugar for function definitions and their equivalents:

function `f` **( ) ... end**	`f` **= function ( ) ... end**
function `a.b.f` **( ) ... end**	`a.b.f` **= function ( ) ... end**
local function `f` **( ) ... end**	**local** `f` **;** `f` **= function ( ) ... end**
function `a.b:f` **(...) ... end**	`a.b.f` **= function (** `self`**, ...) ... end**

When Lua pre-compiles a chunk, all its function bodies are pre-compiled too. Whenever Lua executes the function definition, the function is *instantiated* (or *closed*). This instance (or *closure*) is the final value of the expression. Different instances of the same function may refer to different external local variables and different environment tables.

An adjustment is made to the argument list if required. Parameters act as local variables that are initialized with the argument values. Results are returned using the **return** statement.

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

For example, if there are no extra arguments, `arg` is `{n=0}`. If the extra arguments are 4 and 2, then `arg` is `{4, 2; n=2}`.

10. Visibility

Lua is a lexically scoped. The scope of variables begins at the first statement *after* their declaration and lasts until the end of the innermost block that includes the declaration. Global variables work as expected.

Local variables can be freely accessed by functions defined inside their scope. A local variable used by an inner function is called an *upvalue*, or *external local variable*, inside the inner function. Variables of the same name in an inner scope has precedence. Each instance of an anonymous function (or closures) defines new instances of local variables.

11. Metatables

Every table and userdata object in Lua may have a *metatable* that defines its behavior for certain operations. An object's behavior can be changed for some operations by setting specific fields in its metatable.

Keys in a metatable are called *events* and the values (functions), *metamethods.* Query metatables with **getmetatable** and change them with **setmetatable**.

Behavior

When Lua performs a metamethod-associated operation, it checks whether that object has a metatable with the corresponding event. If so, the value associated with that key is used.

The key for each operation is a string with its name prefixed by two underscores, for instance, the key for operation "add" is the string "`__add`".

Operations

The following is a *simplified* pseudo code form of operations semantics:

add

the `+` *operation*
If (both are numeric) do `return o1 + o2`
Get handler: `h = getbinhandler(op1, op2, "__add")`
If (handler defined) do `return h(op1, op2)`
If (no handler) call `error("...")`

sub *the* `-` *operation, similar to the* **add** *operation*
mul *the* `*` *operation, similar to the* **add** *operation*
div *the* `/` *operation, similar to the* **add** *operation*

For `getbinhandler`, Lua tries to get the handler from the first operand, then it tries the second operand. For `getcomphandler`, both objects has to be of the same type, using the same metamethod for the selected operation.

pow

the `^` *(exponentiation) operation*
If (both are numeric) call `return __pow(o1, o2)`
Get handler: `h = getbinhandler(op1, op2, "__pow")`
If (handler defined) do `return h(op1, op2)`
If (no handler) call `error("...")`

umn

the unary `-` *operation*
If (numeric) do `return -o`
Get handler: `h = metatable(op).__unm`
If (handler defined) do `return h(op, nil)`
If (no handler) call `error("...")`

concat

the `..` *(concatenation) operation*
If (both string or numeric) do `return op1 .. op2`
Get handler: `h = getbinhandler(op1, op2, "__concat")`
If (handler defined) do `return h(op1, op2)`
If (no handler) call `error("...")`

eq

the `==` *operation*
If (different types) return false
If (`op1 == op2`) return true
Get handler: `h = getcomphandler(op1, op2, "__eq")`
If (handler defined) do `return h(op1, op2)`
If (no handler) return false

lt

the `<` *operation*
If (both numeric) do numeric `return op1 < op2`
If (both string) do lexicographic `return op1 < op2`
Get handler: `h = getcomphandler(op1, op2, "__lt")`
If (handler defined) do `return h(op1, op2)`
If (no handler) call `error("...")`

le

the `<=` *operation*
If (both numeric) do numeric `return op1 <= op2`
If (both string) do lexicographic `return op1 <= op2`
Get handler: `h = getcomphandler(op1, op2, "__le")`
If (handler defined) do `return h(op1, op2)`
If (no handler) get handler for `"__lt"`
  If (handler defined) do `return not h(op2, op1)`
  If (no handler) call `error("...")`

`a~=b` is equivalent to `not(a==b)`; `a>b` is equivalent to `b<a`; `a>=b` is equivalent to `b<=a`. In the absence of a **le** metamethod, Lua tries **lt**, assuming that `a<=b` is equivalent to `not(b<a)`.

index

the indexing access `table[key]` *(*`gettable` *event)*
If (object is a table)
  Get raw value: `v = rawget(table, key)`
  If (`v` is not **nil**) do `return v`
  Get handler: `h = metatable(table).__index`
  If (no handler) do `return nil`
If (object is not table)
  Get handler: `h = metatable(table).__index`
  If (no handler) call `error("...")`
If (handler is a function) do `return h(table, key)`
  Else do `return h[key]` (repeat)

newindex

the indexing assignment `table[key] = value` *(*`settable` *event)*
If (object is a table)
Get raw value: `v = rawget(table, key)`
If (`v` is not **nil**) do `rawset(table, key, value); return`
Get handler: `h = metatable(table).__newindex`
If (no handler) do `rawset(table, key, value); return`
If (object is not table)
Get handler: `h = metatable(table).__newindex`
If (no handler) call `error("...")`
If (handler is a function) do `return h(table, key, value)`
Else do `h[key] = value` (repeat)

call

called when Lua calls a value (`function` *event)*
If (object is a function) do `return func(unpack(arg))`
If (object is not a function)
Get handler: `h = metatable(func).__call`
If (handler defined) do `return h(func, unpack(arg))`
If (no handler) call `error("...")`

Other keys (detailed elsewhere) are: "`__pow`" (global), "`__gc`", "`__mode`", "`__fenv`", "`__metatable`", and "`__tostring`".

12. Garbage Collection

Lua runs a *garbage collector* (GC) from time to time to collect all *dead objects.* All objects in Lua are subject to automatic management.

Lua uses two control numbers: the byte counter counts the amount of dynamic memory in use; the other is a threshold. When the number of bytes crosses the threshold, Lua runs the GC. The byte counter is adjusted, and then the threshold is reset to twice the new value of the byte counter.

Garbage-Collection Metamethods

You can set GC metamethods for userdata (*finalizers*), to coordinate Lua's GC with external resource management. Free userdata with a field `__gc` in their metatables are not collected immediately.

At the end of each GC cycle, finalizers for userdata are called in *reverse* order of their creation, among those collected in that cycle. (First finalizer called was the last one created.)

Weak Tables

A *weak table* is a table whose elements are *weak references.* If the only references to an object are weak references, then the GC will collect that object. A weak table can have weak keys, weak values, or both. If either the key or the value is collected, the whole pair is removed.

The weakness of a table is controlled by `__mode` in its metatable. If the field is a string containing character `k`, keys are weak. `v` denotes weak values.

A table used as a metatable should not have its `__mode` changed, otherwise the weak behavior of the tables controlled by this metatable is undefined.

13. Coroutines

Coroutines represents independent threads of execution. A coroutine suspend execution by explicitly yielding (collaborative multithreading.)

- Created by calling **coroutine.create**, passing the coroutine function (no execution). A handle (object type *thread*) is returned.
- Executed by calling **coroutine.resume**, passing the handle and arguments.
- Coroutine executes until it terminates (via a normal return or an error) or *yields* by calling **coroutine.yield** plus optional arguments.
- **coroutine.resume** normally returns **true**, plus any values returned by the coroutine, or **false** plus an error message.
- When execution resumes, **coroutine.yield** returns the extra arguments that were passed to **coroutine.resume**.
- **coroutine.wrap** creates an alternate coroutine form (see the Basic Library.)

14. The Lua C API

The Lua C API is declared in `lua.h`. API functions implemented as macros uses each argument exactly once and do not generate hidden side-effects.

Lua States

Lua is fully reentrant: it has no global variables. The whole state is stored in a dynamically allocated structure of type `lua_State`.

`lua_State *`**`lua_open`**` (void);`

Creates a state. Returns a pointer to its `lua_State` structure.

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

Releases a state. Destroys all objects, frees all dynamic memory. Optional (usually all resources are released when a program ends.)

15. Stack API Functions

Whenever Lua calls C, the called function gets a new, independent, stack that initially contains any arguments to the C function. The C function pushes its results to be returned to the caller on the same stack. Lua ensures that at least `LUA_MINSTACK` stack positions are available (usually defined as 20.)

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, the *valid* index values are:

`n`	`-1`	last element	top of stack
`1`	`-n`	first element	bottom of stack

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 <= stackspace)
```

Most functions accepts *pseudo-indices* as well, for non-stack Lua values.

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

Index of the top element, also the number of elements in the stack.

```
int lua_checkstack (lua_State *L, int extra);
```

Grows the stack size to `top+extra` elements; returns false if it fails to do so. Never shrinks the stack.

Stack Manipulation

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

`lua_settop` accepts any acceptable index, or 0, and sets the stack top to that index. If new top > old top, new elements are filled with **nil**. If `index` is 0, then all stack elements are removed. `lua_pushvalue` pushes onto the stack a copy of the element at the given index. `lua_insert` moves the top element into the given position, shifting up elements.

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

`lua_remove` removes the element at the given position, shifting down elements. `lua_replace` moves the top element into the given position, without shifting any element (therefore replacing the value).

In addition, **`lua_pop`**`(L,n)` is a macro which pops `n` elements from the stack. All these functions accept only valid indices.

Querying the Stack

To check the type of a stack element, the following functions (which can be called with any acceptable index) 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_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_THREAD`, and `LUA_TLIGHTUSERDATA`. Returns `LUA_TNONE` if index is non-valid.

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

Translates type constants to strings. `"no value"` if index is non-valid.

```
int lua_isnil           (lua_State *L, int index);
int lua_isboolean       (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);
int lua_islightuserdata (lua_State *L, int index);
```

Returns 1 if the object is compatible with the given type, 0 otherwise. `lua_isboolean` is an exception: It succeeds only for boolean values. Always returns 0 for a non-valid index.

`lua_isnumber` and `lua_isstring` accepts numbers and strings (coercion). `lua_isfunction` accepts both Lua and C functions. `lua_isuserdata` accepts both full and light userdata.

To distinguish between two types, you must call a different function.

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

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

`lua_equal` and `lua_lessthan` are equivalent to **==** and **<** in Lua. `lua_rawequal` compares for equality, without metamethods. Returns 0 if any indices are non-valid.

Getting Values from the Stack

These functions accepts any acceptable index. An invalid index gives the same result as an incorrect type (returns 0 or `NULL`).

```
int        lua_toboolean (lua_State *L, int index);
lua_Number lua_tonumber  (lua_State *L, int index);
const char *lua_tostring (lua_State *L, int index);
size_t     lua_strlen    (lua_State *L, int index);
```

By default, `lua_Number` is `double`. `lua_toboolean` gives 0 (for **false** or **nil**) or 1. `lua_tonumber` and `lua_tostring` follow coercion rules. `lua_tostring` may *change* a number in the stack to a *string*. Strings are null-terminated, may have embedded zeros, and are subject to GC (use the registry to avoid GC.)

```
lua_CFunction lua_tocfunction (lua_State *L, int index);
void          *lua_touserdata (lua_State *L, int index);
lua_State     *lua_tothread   (lua_State *L, int index);
void          *lua_topointer  (lua_State *L, int index);
```

These functions returns `NULL` if the value's type is invalid. `lua_topointer` converts a userdata, table, thread, or function value to a generic C pointer (`void *`). Different objects of the same type return different pointers. The process is not directly reversible.

Pushing Values onto the Stack

```
void lua_pushboolean   (lua_State *L, int b);
void lua_pushnumber    (lua_State *L, lua_Number 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);
void lua_pushlightuserdata (lua_State *L, void *p);
```

These functions receive a C value, convert it to a corresponding Lua value, and push the result onto the stack. `lua_pushstring` accepts only proper C strings; `lua_pushlstring` accepts strings with an explicit size. An *internal copy* of a given string is made.

```
const char *lua_pushfstring  (lua_State *L,
                              const char *fmt, ...);
const char *lua_pushvfstring (lua_State *L, const char
                              *fmt, va_list argp);
```

`sprintf`- and `vsprintf`-style formatted strings. Lua handles memory allocation. The only valid specifiers are: `%%`, `%s`, `%f`, `%d`, `%c`.

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

Concatenates using Lua semantics `n` values at the top of the stack, pops them, and leaves the result at the top. 0 results in an empty string.

16. Miscellaneous API Functions

Garbage Collection

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

Returns either the byte counter value or the threshold value (in Kbytes).

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

Sets the new threshold value in Kbytes. If new threshold < byte counter, then Lua immediately runs the GC. After collection, a new threshold is set according to the usual rule. A 0 value forces a GC.

Userdata

Userdata represents C values in Lua. Lua supports two types of userdata. The kind of userdata can only be tested in C.

- A *full userdata* represents a block of memory (an object); it must be created, can have a metatable, can be detected during collection; equal only to itself.
- A *light userdata* represents a pointer; it is a value; it is not created, has no metatables; it is not collected; equality is by comparing pointer addresses.

```
void *lua_newuserdata (lua_State *L, size_t size);
```

Create a new full userdata. Allocates memory, then pushes on the stack a new userdata with the block address, and returns this address.

Use `lua_pushlightuserdata` for light userdata. `lua_touserdata` returns the block address (full); the pointer (light); or **nil**. During collection, Lua calls the userdata's **gc** metamethod, if any, and then it frees the userdata's corresponding memory.

Metatables

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

Pushes the metatable of a given object. If call fails, returns 0 and pushes nothing.

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

Pops a table and sets it as the new metatable for the given object. If call fails, returns 0.

Loading Lua Chunks

```
int lua_load (lua_State *L, lua_Chunkreader reader,
              void *data, const char *chunkname);
typedef const char * (*lua_Chunkreader)
              (lua_State *L, void *data, size_t *size);
```

Loads a chunk. Returns 0, `LUA_ERRSYNTAX` or `LUA_ERRMEM`. Pushes the compiled chunk as a Lua function, or an error message.

Automatically detects text or binary forms. The `data` pointer is passed to `reader`, which returns pieces of chunk data and sets `size` (where `size>0`). To end, the reader returns `NULL`. The reader cannot call any Lua function. `chunkname` is used to identify the chunk.

17. Manipulating Tables and Environments

Tables

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

Creates a new, empty table and pushes it onto the stack.

```
void lua_gettable (lua_State *L, int index);
void lua_rawget   (lua_State *L, int index);
```

`index` points to the table. Pops a key from the stack and pushes the contents of the table at that key. `lua_gettable` may trigger a metamethod for the **index** event. `lua_rawget` avoids invoking metamethods.

```
void lua_settable (lua_State *L, int index);
void lua_rawset   (lua_State *L, int index);
```

To store a value into a table: (1) push key, (2) push value, and (3) make call. `index` points to the table. Pops both the key and the value. May trigger a metamethod for the **settable** or **newindex** events. `lua_rawset` avoids invoking metamethods.

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

Traverse a table pointed to by `index`. Pops a key, and pushes a key-value pair from the table to the stack (the "next" pair after the given key.) A **nil** key signals the start. At the end, `lua_next` returns 0 and pushes nothing. Typically:

```
lua_pushnil(L);                        /* initialize */
while (lua_next(L, t) != 0) {...}      /* loop */
```

Usually a `lua_pop(L, 1)` is done to remove the value, keeping the key for the next iteration. Note that `lua_tostring` might convert the key in-place, do not use unless the key is already a string.

Environments

All global variables are kept in ordinary Lua tables, called environments. The initial environment is called the global environment, held at the pseudo-index `LUA_GLOBALSINDEX`. To access and change globals, use regular table operations over an environment table. `lua_replace` can change the global environment of a Lua thread.

```
void lua_getfenv (lua_State *L, int index);
int  lua_setfenv (lua_State *L, int index);
```

lua_getfenv pushes on the stack the environment table of the given function. For C functions, it pushes the global environment.

lua_setfenv pops a table from the stack and sets it as the new environment for the given function. Returns 0 if object not a function.

Using Tables as Arrays

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

Helps use Lua tables as arrays, indexing with numbers only. n is the *n*-th element of the table at position index. lua_rawgeti pushes to the stack while lua_rawseti pops from the stack.

18. Manipulating Functions

Calling Functions

Functions defined in Lua and C functions registered in Lua can be called from the host by: (a) push the function to be called, (b) push the function's arguments in *direct order*, (c) make the call using lua_call.

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

nargs is the number of arguments pushed (direct order). Everything is popped, and the function results are pushed (direct order). The number of results are adjusted to nresults, unless it is LUA_MULTRET (all results pushed). Macro are often used to simplify calls.

Protected Calls

With lua_call, any error inside the called function is propagated upwards (with a longjmp). You can handle errors with *protected calls.*

```
int lua_pcall (lua_State *L, int nargs,
               int nresults, int errfunc);
```

If no errors occur, lua_pcall is exactly like lua_call. However, if there is any error, lua_pcall catches it, pushes a single value (the error message), and returns an error code.

errfunc may specify the valid stack index of an *error handler function* that will handle additional error message processing.

lua_pcall returns 0 if successful or LUA_ERRRUN (a runtime error), LUA_ERRMEM (a memory allocation error, errfunc is not called), or LUA_ERRERR (a error while running errfunc).

Defining C Functions

Lua can be extended with functions written in C, which must be of type:

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

```
lua_register (lua_State *L, const char *name,
              lua_CFunction fn);
```

Convenience macro to register a C function to Lua.

- A C function receives a Lua state and returns a number.
- Arguments are received in its stack in direct order.
- When the function starts, its first argument is at index 1.
- `lua_gettop(L)` gives the number of arguments.
- Push return values in direct order and return the number of results.
- Other stack values below the results will be discharged by Lua.

Defining C Closures

When some values are associated with a C function, a *C closure* is created. First push the values onto the stack (direct order). Then call:

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

Pushes the C function onto the stack. `n` is the number of values associated with the function, which are then popped. The values are located at specific pseudo-indices when the function is called. Use the macro `lua_upvalueindex(i)` where the first value is `lua_upvalueindex(1)`, and so on. An out of range index is acceptable but invalid.

Registry

A registry is a predefined table that can be used by any C code to store whatever Lua value it needs to store, so that they survive outside the life span of a C function. The registry is located at pseudo-index `LUA_REGISTRYINDEX`.

Typically, a library should use as key a string containing its name, or a light userdata with the address of a C object in the code. The integer keys in the registry are used by the reference mechanism, implemented by the auxiliary library, and therefore should not be used by other purposes.

Error Handling in C

Lua uses the C `longjmp` facility to handle errors. Lua *raises* an error by doing a long jump. A *protected environment* uses `setjmp` to set a recover point; any error jumps to the most recent active recover point. Outside any protected environment, Lua calls a *panic function* and then calls `exit(EXIT_FAILURE)`.

```
lua_CFunction lua_atpanic (lua_State *L,
                           lua_CFunction panicf);
```

The new panic function may avoid application exit by never returning but the Lua state will not be consistent and should be closed. `lua_open`, `lua_close`, `lua_load`, and `lua_pcall` run in protected mode so they never raise an error.

```
int lua_cpcall (lua_State *L, lua_CFunction func,
                void *ud);
```

Runs a given C function in protected mode. `func` starts with only one element in its stack, a light userdata containing `ud`. In case of errors, `lua_cpcall` acts like `lua_pcall`, plus the error object on the top of the stack; otherwise, it returns 0, with the stack unchanged. Return values of `func` are discarded.

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

Generate a Lua error in C code. The error message (or object) must be on the top of the stack. `lua_error` does a long jump, never returns.

19. Threads

Lua offers partial support for multiple threads of execution. Coroutines are implemented on top of threads.

```
lua_State *lua_newthread (lua_State *L);
```

Creates a new thread. Pushes the thread on the stack and returns a `lua_State` pointer to represent this new thread. Shares initially global objects with `L`, but has an independent stack. The global environment table can be changed independently for each thread.

There is no explicit function to close or to destroy a thread. Threads are subject to garbage collection, like any Lua object.

```
int lua_resume (lua_State *L, int narg);
int lua_yield  (lua_State *L, int nresults);
```

Manipulate threads as coroutines. To run a coroutine: (a) push the body function, (b) push arguments, (c) then call `lua_resume` with the number of arguments `narg`. Upon return, the stack contains all values returned (passed to `lua_yield`). `lua_resume` returns 0, or an error code plus an error message on the stack.

`lua_yield` can only be called as the return expression of a C function. `nresults` is the number of values on the stack that are passed as results to `lua_resume`.

```
void lua_xmove (lua_State *from, lua_State *to, int n);
```

Exchanges values. Pops `n` values from the stack `from`, and pushes them into the stack `to`.

20. The Debug Interface

Stack and Function Information

The structure `lua_Debug` is used to carry information about an active function:

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

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

Fills the private parts of a `lua_Debug` structure `ar` with an *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*. Returns 1; if level is greater than stack depth, returns 0.

```
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. For example '`l`' fills in `currentline`. Moreover, '`f`' pushes onto the stack the function that is running at the given level.

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

`source`	A string where the function was defined, or if it was from a file, an '`@`' character followed by the file name.
`short_src`	"Printable" version of `source`, for error messages.
`linedefined`	Line number where the definition of the function starts.
`what`	"`Lua`" for a Lua function, "`C`" for a C function, "`main`" for the main part of a chunk, or "`tail`" for a tail call.
`currentline`	Current line where the given function is executing. When no line information is available, this is set to -1.
`name`	A reasonable name, otherwise it is set to `NULL`.
`namewhat`	Explains `name`. Can be "`global`", "`local`", "`method`", "`field`", or "", according to how the function was called.
`nups`	Number of upvalues of the function.

Manipulating Local Variables and Upvalues

```
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);
```

The first parameter or local variable has index 1, and so on, until the last active local variable. Upvalues have no particular order. `ar` must be valid, filled by `lua_getstack` or the hook mechanism.

`lua_getlocal` gets the index of a local variable (`n`), pushes its value onto the stack, and returns its name. `lua_setlocal` assigns the value at the top of the stack to the variable and returns its name. Both return `NULL` when the index is out of range.

```
const char *lua_getupvalue (lua_State *L,
                            int funcindex, int n);
const char *lua_setupvalue (lua_State *L,
                            int funcindex, int n);
```

The upvalues of a function are accessible even when the function is not active. The functions operate on both Lua (external local variables that are included in its closure) and C functions. `funcindex` points to a function in the stack.

`lua_getpuvalue` gets the index `n` of an upvalue, pushes its value onto the stack, and returns its name. `lua_setupvalue` assigns the value at the top of the stack to the upvalue and returns its name. Both return `NULL` when the index is out of range. For C functions, all upvalues have an empty string as a name.

Hooks

Hooks are user-defined C functions that are called during the program execution, in four different events: *call* (`LUA_HOOKCALL`), *return* (`LUA_HOOKRET` or `LUA_HOOKTAILRET`), *line* (`LUA_HOOKLINE`), and *count* (`LUA_HOOKCOUNT`).

```
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);
int lua_sethook (lua_State *L, lua_Hook func,
                 int mask, int count);
```

Sets debugging hooks. `mask` is specified by a disjunction of the constants `LUA_MASKCALL`, `LUA_MASKRET`, `LUA_MASKLINE`, and `LUA_MASKCOUNT`. `count` is only meaningful for `LUA_MASKCOUNT`. A hook is disabled by setting `mask` to zero.

call hook	Called just after Lua enters the new function.
return hook	Called just before Lua leaves the function.
line hook	Called a new line of code is about to be executed, or when it jumps back in the code (even to the same line.) (Lua functions only.)
count hook	Called every `count` instructions. (Lua functions only.)

```
lua_Hook lua_gethook      (lua_State *L);
int      lua_gethookmask  (lua_State *L);
int      lua_gethookcount (lua_State *L);
```

Gets the current hook, the current mask, or the current count.

Whenever a hook is called, its `ar` argument has its `event` field set to the specific event that triggered the hook. For line events, the `currentline` field is also set. To get the value of any other field, the hook must call `lua_getinfo`. Return events may be `LUA_HOOKRET` or `LUA_HOOKTAILRET`.

While Lua is running a hook, it disables other calls to hooks. Therefore, if a hook calls back Lua to execute a function or a chunk, that execution occurs without any calls to hooks.

21. Standard Libraries

Except for the basic library, each library provides all its functions as fields of a global table or as methods of its objects. The initialization functions are (declared in header file `lualib.h`):

basic library	`int luaopen_base (lua_State *L);`
string library	`int luaopen_string (lua_State *L);`
table library	`int luaopen_table (lua_State *L);`
mathematical library	`int luaopen_math (lua_State *L);`
I/O and OS libraries	`int luaopen_io (lua_State *L);`
debug library	`int luaopen_debug (lua_State *L);`

22. Basic Function Library

assert (V [, MESSAGE])

Error when V is **nil** or **false**, otherwise returns this value. MESSAGE is an error message, defaults to "assertion failed!".

collectgarbage ([LIMIT])

Sets the GC threshold to LIMIT (KB) and checks it against the byte counter. If new threshold < byte counter, immediately runs the GC. Default 0 (forced GC cycle.)

dofile (FILENAME)

Opens FILENAME and executes it as a Lua chunk. Default is `stdin`. Returns any value returned by the chunk. Propagates errors.

error (MESSAGE [, LEVEL])

Terminates the last protected function called, returns MESSAGE as the error message. Never returns. For LEVEL 1 (default) the error position pointed to is where **error** was called; 2 gives the parent, etc.

_G

Global; holds the global environment (`_G._G = _G`). Changing **_G** does not affect any environment. (**setfenv** changes environments.)

getfenv (F)

Returns the current environment in use by the function. F can be a function, or a stack level number. Level 1 (default) is the function calling **getfenv**. If non-Lua function or F is 0, the global environment is returned. An "`__fenv`" environment field overrides the normal return value.

getmetatable (OBJECT)

Returns **nil** if no metatable, else if the object's metatable has a "`__metatable`" field, returns the associated value, else returns the metatable of the object.

gcinfo ()

Returns two results: (1) KB of dynamic memory in use, and (2) the current GC threshold (KB).

ipairs (T)

Returns an iterator function, the table T, and 0, for use as the **in** expression in a generic **for** construction: `for i,v in ipairs(t)`…

loadfile (FILENAME)

Loads a file as a chunk. Compile-only. Does not run. Returns the compiled chunk as a function; otherwise, returns **nil** plus error message. Environment of the returned function is the global environment.

loadlib (LIBNAME, FUNCNAME)

Links in the dynamic C library LIBNAME. Returns FUNCNAME as a C function. A proper path must be specified for LIBNAME. Non ANSI C. Uses the `dlfcn` standard.

loadstring (STRING [, CHUNKNAME])

Loads a string as a Lua chunk. Does not run. Returns the compiled chunk as a function; otherwise, returns **nil** plus error message. The returned function uses the global environment. CHUNKNAME is the optional debug name. Recommended idiom: `assert(loadstring(s))()`

next (TABLE [, INDEX])

Traverse all fields of a table. Returns the next index, value pair. If INDEX is **nil** (the default), starts with the first index. When called with the last index, or with **nil** in an empty table, **next** returns **nil**.

Only fields with non-**nil** values are considered. Enumeration order is not specified. For numeric order, use a numerical **for** or the **ipairs** function. Behavior is *undefined* if the table is changed during the traversal.

pairs (T)

Returns the **next** function and the table `T` plus a **nil**, for use in a generic **for** construction: `for k,v in pairs(t) do ... end`

pcall (F, ARG1, ARG2, …)

Calls function `F` with the given arguments in protected mode. **pcall** catches any errors and returns a status code. Returns **true** plus return results if success, or **false** plus the error message if error.

print (E1, E2, …)

Prints arguments to `stdout` using strings returned by **tostring**. Not intended for formatted output; typically for debugging.

rawequal (V1, V2)

Equality check; returns a boolean, without invoking any metamethod.

rawget (TABLE, INDEX)

Gets real value of `table[index]`, without invoking metamethods. INDEX should not be **nil**.

rawset (TABLE, INDEX, VALUE)

Sets the real value of `table[index]` to VALUE, without invoking any metamethod. TABLE must be a table, and INDEX must be non-**nil**.

require (PACKAGENAME)

Loads the given package. Checks table `_LOADED` first. If loaded, **require** returns the value returned during the first loading. Otherwise, it searches a path for a file: (a) global string `LUA_PATH`, (b) environment variable `LUA_PATH`, and (c) "`?;?.lua`". Lua inserts PACKAGENAME in place of the "?" for each *template.*

The package name is associated in table `_LOADED` with the return value, which is returned. A return value of **nil** (or no value) is converted to **true**. A package may be reloaded if **false**. May signal an error. Global `_REQUIREDNAME` defined with the package name.

setfenv (F, TABLE)

Sets the current environment to be used by F, which can be a function or a stack level. When F is 0, the global environment of the running thread is changed. If the original environment has a "`__fenv`", an error is raised.

setmetatable (TABLE, METATABLE)

Sets the metatable for TABLE. A **nil** removes the metatable. If the original metatable has a "`__metatable`", an error is raised.

tonumber (E [, BASE])

Converts E to a number using optional BASE, **nil** if unsuccessful. BASE valid from 2 to 36. Digits are `[0-9A-Z]` (case insensitive). In decimal, a fraction and exponent is optional. Other bases must be unsigned.

tostring (E)

Converts E to a string in a reasonable format. See also **format**. If E has a "`__tostring`" metatable field, that metamethod is used instead.

type (V)

Returns the type as a string, one of: `"nil"`, `"number"`, `"string"`, `"boolean"`, `"table"`, `"function"`, `"thread"`, and `"userdata"`.

unpack (LIST)

Returns all elements from the given list. This function is equivalent to: `return list[1], list[2], ..., list[n]`

_VERSION

A global that holds the current interpreter version (`"Lua 5.0"`).

xpcall (F, ERR)

Calls F in protected mode, with ERR as the error handler. Any error is caught, and ERR is called. Return results are similar to **pcall**, except it returns a false with the result from ERR.

Coroutine Manipulation

coroutine.create (F)

Creates a new coroutine, with body F. Returns its thread object.

coroutine.resume (CO, VAL1, …)

Starts or continues execution of coroutine CO. Other arguments are passed to the body function or as the results from the yield. If successful, returns **true** plus any values passed to **yield** or returned by the body function, otherwise returns **false** plus an error message.

coroutine.status (CO)

Returns the status of CO: `"running"`, `"suspended"`, or `"dead"`.

coroutine.wrap (F)

Creates new coroutine with body F. Returns a function that resumes coroutine. Does not return boolean status. Propagates errors.

coroutine.yield (VAL1, …)

Suspends execution of coroutine, which cannot be running a C function, a metamethod, or an iterator. Extra arguments go as results to **resume**.

23. String Manipulation Library

The first character is at *position 1* (not at 0). Negative indices are for backwards indexing (e.g. the last character is at position *-1*.)

string.byte (S [, I])

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

string.char (I1, I2, …)

Receives 0 or more integers and returns a string with corresponding characters of the equivalent numerical code. Not portable.

string.dump (FUNCTION)

Returns a binary representation of FUNCTION, which must be a Lua function without upvalues. See **loadstring**.

string.find (S, PATTERN [, INIT [, PLAIN]])

Looks for the first *match* of PATTERN in S. If it finds one, returns the start and end indices; otherwise, returns **nil**. Captures are returned as extra results. INIT optionally specifies where to start, defaults to 1, may be negative. If PLAIN is 1, pattern matching facility is turned off.

string.len (S)

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

string.lower (S)

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

string.rep (S, N)

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

string.sub (S, I [, J])

Returns substring of S, starting at I and running until J. Indices may be negative. J defaults to -1 (string length.) Also for prefix and suffix.

string.upper (S)

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

string.format (FORMATSTRING, E1, E2, …)

Similar to `printf`. Returns formatted version of E1, E2, … using the given FORMATSTRING description. `*`, `l`, `L`, `n`, `p`, and `h` are not supported. `q` formats a string with suitable escapes to be safely read back by Lua.

The options `c`, `d`, `E`, `e`, `f`, `g`, `G`, `i`, `o`, `u`, `X`, and `x` all expect a number as argument. `q` and `s` expect a string. The `*` modifier must be simulated. `%s` strings cannot contain embedded zeros.

string.gfind (S, PAT)

Returns an iterator function for returning the next captures from pattern PAT over string S. If PAT specifies no captures, then the whole match is produced. For generic **for** loops.

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

Returns: (1) a copy of S with all PAT patterns replaced by string REPL, plus (2) the total substitutions made. N limits substitutions.

If REPL is a string, then its value is used for replacement. %n (1<=n<=9) sequences refers to the *n*-th captured substring, which will be substituted in. If REPL is a function, then it is called with all captured substrings (or the whole match) passed as arguments. A returned string result is used as the replacement, else the replacement is an empty string.

24. Table Manipulation Library

A table's size can be: (a) the field "n" if it is numeric, or (b) the value explicitly set using **table.setn**, or (c) one less the first integer index with a **nil** value.

table.concat (TABLE [, SEP [, I [, J]]])

Returns a concatenation of table elements I to J with separator SEP. I defaults to 1 and J defaults to table size. SEP is empty by default.

table.foreach (TABLE, F)

Executes function F over all elements of TABLE. F is called with each index and value pair. If F returns a non-**nil** value, the loop is broken, and this value is returned as the final value.

table.foreachi (TABLE, F)

Similar to **table.foreach** except it is for numerical indices (1 to n).

table.getn (TABLE)

Returns size of a table seen as a list, using the usual rules.

table.sort (TABLE [, COMP])

Sorts table elements, *in-place*, from index 1 to n. Optional COMP must be a function, receives 2 elements, returns true when first < second. Defaults to operator **<**. The sort algorithm is *not* stable.

table.insert (TABLE, [POS,] VALUE)

Inserts element VALUE at POS in TABLE, shifting up to open space if necessary. POS defaults to n+1 (append). Updates table size using **table.setn**.

table.remove (TABLE [, POS])

Removes from TABLE element at POS, shifting down to close the space if necessary. Returns element's value. POS defaults to n (last element removed). Updates table size using **table.setn**.

table.setn (TABLE, N)

Updates the size of a table. Updates field "n", or an internal state.

25. Mathematical Function Library

Similar to standard C math. A **math.pi** is provided, and a global __**pow** is also registered for the operator **^**. Trigonometric functions uses radians.

math.abs (V)	absolute	**math.frexp** (V)	mantissa, exp
math.acos (V)	arc cosine	**math.ldexp** (V1, V2)	v1*2^v2
math.asin (V)	arc sine	**math.log** (V)	natural log
math.atan (V)	arc tangent	**math.log10** (V)	log 10
math.atan2 (V1, V2)	arc tan v1/v2	**math.mod** (V1, V2)	modulus v1/v2
math.ceil (V)	smallest int >= v	**math.pow** (V1, V2)	v1^v2
math.cos (RAD)	cosine	**math.rad** (DEG)	deg to rad
math.deg (RAD)	rad to deg	**math.sin** (RAD)	sine
math.exp (V)	e^v	**math.sqrt** (V)	square root
math.floor (V)	largest int <= v	**math.tan** (RAD)	tangent

math.max (V1, …)
math.min (V1, …)

Returns the maximum or minimum in a list of one or more values.

math.random ([N [, U]])
math.randomseed (SEED)

math.random returns a real in the range [0,1) with no arguments. With a number N, returns an integer in the range [1,n]. With two arguments, returns an integer in the range [l,u]. **math.randomseed** sets a seed for the pseudo-random generator.

26. I/O and OS Facilities

Implicit file operations are supplied by table **io**. Explicit file operations are methods of an explicit file descriptor returned by a call to **io.open**.

The predefined file descriptors are: **io.stdin**, **io.stdout**, and **io.stderr**. A file handle is a userdata containing the file stream (`FILE*`), with a metatable created by the I/O library. Most I/O functions return **nil** on failure plus an error message, or some non-**nil** value on success.

io.close ([FILE])

Equivalent to `file:close()`. Without FILE, closes default output file.

io.flush ()

Equivalent to `file:flush` over the default output file.

io.input ([FILE])

Opens FILE (in text mode) and sets its handle as the default input file; sets a file handle as the default; or returns current default. Raises errors.

io.lines ([FILENAME])

Opens FILENAME in read mode, returns a **for** iterator function that read the file line-by-line. Returns **nil** if end of file, and closes it. Without FILENAME, uses the default input file.

io.open (FILENAME [, MODE])

Opens a file in the MODE specified. Returns a new file handle.

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		

io.output ([FILE])

Similar to `io.input`, but operates over the default output file.

io.read (FORMAT1, …)

Equivalent to `io.input():read`.

io.tmpfile ()

Returns a handle for a temporary file, opened in update mode. Automatically removed when the program ends.

io.type (OBJ)

Returns the `"file"` if OBJ is an open file handle, `"closed file"` if closed, and **nil** if it is not a file handle.

io.write (VALUE1, …)

Equivalent to `io.output():write`.

file:close ()

Closes `file`.

file:flush ()

Saves any written data to `file`.

file:lines ()

Returns an iterator function that reads the file line-by-line. Does not close the file when the loop ends.

file:read (FORMAT1, …)

Reads FILE according to the given formats. Each format returns a string or a number, or **nil** if it fails. The formats are:

`*n`	reads a number, and returns a number
`*a`	reads the whole file, starting at the current position. On EOF, it returns an empty string
`*l`	(default) reads next line (EOL skipped), or **nil** on EOF
`number`	reads a string with up to that number of characters, or **nil** on EOF. If 0, reads nothing and returns empty string.

file:seek ([WHENCE] [, OFFSET])

Sets and gets the file position, to the position given by OFFSET from a base specified by WHENCE, where:

`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 absolute file position. On error, returns **nil**, plus an error message. Default for WHENCE is `cur`; default OFFSET is 0. `file:seek()` returns the current file position.

file:write (VALUE1, …)

Writes arguments to filehandle `file`. Must be strings or numbers.

os.clock ()

Returns approximate amount of CPU time used by the program (sec).

os.date ([FORMAT [, TIME]])

Returns string with date and time formatted according to FORMAT. Default to current time. Uses `strftime` rules (default "`%c`"); "`!`" gives UTC, "`*t`" gives a table: `year` (YYYY), `month` (1–12), `day` (1–31), `hour`, `min`, `sec`, `wday` (Sun=1), `yday`, `isdst` (boolean).

os.difftime (T2, T1)

Returns number of seconds from time T1 to time T2.

os.execute (COMMAND)

Passes COMMAND to be executed by an OS shell. Returns a status code.

os.exit ([CODE])

Terminate the host program. Default is the success code.

os.getenv (VARNAME)

Returns environment variable VARNAME, or **nil** if undefined.

os.remove (FILENAME)

Deletes FILENAME. If fails, returns **nil** plus error message.

os.rename (OLDNAME, NEWNAME)

Renames a file. If fails, returns **nil** plus error message.

os.setlocale (LOCALE [, CATEGORY])

Sets current locale. CATEGORY is an optional string, one of: `"all"` (default), `"collate"`, `"ctype"`, `"monetary"`, `"numeric"`, or `"time"`. Returns the name of the new locale, or **nil** if invalid.

os.time ([TABLE])

Returns the current time (default), or a time as specified by TABLE (must have `year`, `month`, `day`.) Usually in seconds (from an epoch.)

os.tmpname ()

Returns a string with a name for a temporary file. *Unsafe.*

27. The Reflexive Debug Interface

These are provided for debugging etc., and adversely affect performance. The privacy of local variables may be violated.

debug.debug ()

Enters interactive debugging. A `cont` on a line of its own resumes normal execution. Not lexically nested with any function.

debug.gethook ()

Returns current hook settings: hook function, mask, and count.

debug.getinfo (FUNCTION [, WHAT])

Returns a table with information about a function. FUNCTION can also be a stack level, relative to itself (level 0). **nil** if invalid level. The returned table is similar to that of `lua_getinfo`. Option `f` adds a field `func` with the function itself. By default WHAT gets all information.

debug.getlocal (LEVEL, LOCAL)

Returns name and value of local variable with index LOCAL of the function at stack level LEVEL. Returns **nil** if invalid index, raises an error if LEVEL is out of range.

debug.getupvalue (FUNC, UP)

Returns name and value of upvalue with index UP of function FUNC. Returns **nil** if invalid index.

debug.setlocal (LEVEL, LOCAL, VALUE)

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

debug.setupvalue (FUNC, UP, VALUE)

Assigns VALUE to the upvalue with index UP of function FUNC. Returns **nil** if invalid index.

debug.sethook (HOOK, MASK [, COUNT])

Sets function HOOK as a hook. The string mask may use: "`c`" (call hook), "`r`" (return hook), or "`l`" (line hook). If COUNT > 0, sets a count hook. Without arguments, the hook is turned off.

The hook's first parameter is an event string: `"call"`, `"return"`, `"tail return"`, `"line"` (second param is line number), or `"count"`. Stack level 2 is the running function. (0 is **getinfo**, 1 is the hook)

debug.traceback ([MESSAGE])

Returns a string with a call stack traceback. An optional MESSAGE string is appended to the beginning. Typically used with **xpcall**.

28. Patterns

Character Classes

A *character class* is used to represent a set of characters:

`%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`**).

x	A 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; escapes magic characters and punctuations

[set]	Represents a union class of all characters in `set`. Use a `-` (dash) to specify ranges. `%x` classes may also be used as components. Other characters represent themselves.
[^set]	*Complement* of `set`, where `set` is interpreted as above.

For all single letter classes (`%a`, `%c`, …), the corresponding upper-case letter represents its *complement*. The definitions of letter, space, etc. depend on the current locale. `%l` is more portable than `[a-z]` (may not be equivalent.)

Pattern Items

A *pattern item* may be a single character class, which matches any single character in the class. It can be optionally followed by a suffix:

*****	0 or more repetitions, longest possible sequence
+	1 or more repetitions, longest possible sequence
-	0 or more repetitions, shortest possible sequence
?	0 or 1 occurrence
%n	A substring equal to the *n*-th captured string, for ***n*** between 1 and 9
%bxy	***x*** and ***y*** are distinct; matches strings that start with ***x*** and end with ***y***, where ***x*** and ***y*** are *balanced*. E.g. "**%b()**".

Patterns and Captures

A *pattern* is a sequence of *pattern items* A **^** at the beginning anchors the match at the beginning; a **$** at the end anchors the match at the end.

Sub-patterns enclosed in parentheses; they describe *captures.* When a match succeeds, the substrings of the subject string that match captures are stored (*captured*). Captures are numbered according to their left parentheses, starting from 1. The empty capture "**()**" captures the current string position (a number).

29. Lua Stand-alone

The stand-alone interpreter, `lua`, is console-based and includes all standard libraries plus the reflexive debug interface. Its usage is:

```
lua [options] [script [args]]
```

The options are:

-	executes `stdin` as a file
-e *stat*	executes string *stat*
-l *file*	requires *file*
-i	enters interactive mode after running *script*
-v	prints version information
--	stop handling options

Without arguments, the default is "`lua -v -i`" when `stdin` is a terminal, and "`lua -`" otherwise. The environment variable `LUA_INIT` is checked. If it is a filename, lua executes the file, otherwise, lua executes the string itself.

All remaining arguments are collected in a global table called `arg`. Index 0 holds the script name, index 1 the first argument, etc. Table field `n` is set with the number of arguments. Any arguments (options) before the script name go to negative indices.

If global variable `_PROMPT` is defined as a string, then its value is used as the prompt. `-i` enters interactive mode.

On Unix systems, Lua scripts can be made executable using "`chmod +x`" and the "`#!/usr/local/bin/lua`" form. "`#!/usr/bin/env lua`" is more portable.

30. Incompatibilities with Lua 4.0

Changes in the Language

- The whole tag-method scheme was replaced by metatables.
- Function calls written between parentheses result in exactly one value.
- A function call as the last expression in a list constructor (like `{a,b,f()}`) has all its return values inserted in the list.
- The precedence of **or** is smaller than the precedence of **and**.
- **in**, **false**, and **true** are reserved words.
- The old construction `for k,v in t`, where `t` is a table, is deprecated (although it is still supported). Use `for k,v in pairs(t)` instead.
- When a literal string of the form `[[...]]` starts with a newline, this newline is ignored.
- Upvalues in the form `%var` are obsolete; use external local variables instead.

Changes in the Libraries

- Most library functions now are defined inside tables. There is a compatibility script (`compat.lua`) that redefine most of them as global names.
- In the math library, angles are expressed in radians. With the compatibility script (`compat.lua`), functions still work in degrees.
- The **call** function is deprecated. Use `f(unpack(tab))` instead of `call(f, tab)` for unprotected calls, or the new **pcall** function for protected calls.
- **dofile** do not handle errors, but simply propagates them.
- **dostring** is deprecated. Use **loadstring** instead.
- The **read** option `*w` is obsolete.
- The **format** option `%n$` is obsolete.

Changes in the API

- `lua_open` does not have a stack size as its argument (stacks are dynamic).
- `lua_pushuserdata` is deprecated. Use `lua_newuserdata` or `lua_pushlightuserdata` instead.

31. The Complete Syntax of Lua

chunk → { *stat* [';'] }
block → *chunk*
stat → *varlist1* '=' *explist1*
| *functioncall*
| **do** *block* **end**
| **while** *exp* **do** *block* **end**
| **repeat** *block* **until** *exp*
| **if** *exp* **then** *block*
{ **elseif** *exp* **then** *block* } [**else** *block*] **end**
| **return** [*explist1*]
| **break**
| **for** `Name` '=' *exp* ',' *exp* [',' *exp*] **do** *block* **end**
| **for** `Name` { ',' `Name` } **in** *explist1* **do** *block* **end**
| **function** *funcname funcbody*
| **local function** `Name` *funcbody*
| **local** *namelist* [*init*]
funcname → `Name` { '.' `Name` } [':' `Name`]
varlist1 → *var* { ',' *var* }
var → `Name` | *prefixexp* '[' *exp* ']' | *prefixexp* '.' `Name`
namelist → `Name` { ',' `Name` }
init → '=' *explist1*
explist1 → { *exp* ',' } *exp*
exp → **nil** | **false** | **true** | `Number` | `Literal` | *function*
| *prefixexp* | *tableconstructor* | *exp binop exp* | *unop exp*
prefixexp → *var* | *functioncall* | '(' *exp* ')'
functioncall → *prefixexp args* | *prefixexp* ':' `Name` *args*
args → '(' [*explist1*] ')' | *tableconstructor* | `Literal`
function → **function** *funcbody*
funcbody → '(' [*parlist1*] ')' *block* **end**
parlist1 → `Name` { ',' `Name` } [',' '...'] | '...'
tableconstructor → '{' [*fieldlist*] '}'
fieldlist → *field* { *fieldsep field* } [*fieldsep*]
field → '[' *exp* ']' '=' *exp* | *name* '=' *exp* | *exp*
fieldsep → ',' | ';'
binop → '+' | '-' | '*' | '/' | '^' | '..'
| '<' | '<=' | '>' | '>=' | '==' | '~='
| **and** | **or**
unop → '-' | **not**

This booklet is based on the Lua 5 Reference Manual TEX sources. Updated to conform to Lua 5.0.2 documentation, dated Tue Nov 25 16:08:37 BRST 2003.