Lua 5.2 Reference Manual

by Roberto Ierusalimschy,
Luiz Henrique de Figueiredo,
Waldemar Celes

Contents

1 · Introduction

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Lua is intended to be used as a powerful, lightweight, embeddable scripting language for any program that needs one. Lua is implemented as a library, written in *clean C*, the common subset of Standard C and C++.

Being an extension language, Lua has no notion of a 'main' program: it only works *embedded* in a host client, called the *embedding program* or simply the *host*. The host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. Through the use of C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework. The Lua distribution includes a sample host program called `lua`, which uses the Lua library to offer a complete, standalone Lua interpreter, for interactive or batch use.

Lua is free software, and is provided as usual with no guarantees, as stated in its license. The implementation described in this manual is available at Lua's official web site, `www.lua.org`.

Like any other reference manual, this document is dry in places. For a discussion of the decisions behind the design of Lua, see the technical papers available at Lua's web site. For a detailed introduction to programming in Lua, see Roberto's book, *Programming in Lua*.

2 · Basic Concepts

This section describes the basic concepts of the language.

2.1 · Values and Types

Lua is a *dynamically typed language*. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

All values in Lua are *first-class values*. This means that all values can be stored in variables, passed as arguments to other functions, and returned as results.

There are eight basic types in Lua: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread*, and *table*. *Nil* is the type of the value **nil**, whose main property is to be different from any other value; it usually represents the absence of a useful value. *Boolean* is the type of the values **false** and **true**. Both **nil** and **false** make a condition false; any other value makes it true. *Number* represents real (double-precision floating-point) numbers. Operations on numbers follow the same rules of the underlying C implementation, which, in turn, usually follows the IEEE 754 standard. (It is easy to build Lua interpreters that use other internal representations for numbers, such as single-precision floats or long integers; see file `luaconf.h`.) *String* represents immutable sequences of bytes. Lua is 8-bit clean: strings can contain any 8-bit value, including embedded zeros (`'\0'`).

Lua can call (and manipulate) functions written in Lua and functions written in C (see §3.4.9).

The type *userdata* is provided to allow arbitrary C data to be stored in Lua variables. A userdata value is a pointer to a block of raw memory. There are two kinds of userdata: full userdata, where the block of memory is managed by Lua, and light userdata, where the block of memory is managed by the host. Userdata has no predefined operations in Lua, except assignment and identity test. By using *metatables*, the programmer can define operations for full userdata values (see §2.4). Userdata values cannot be created or modified in Lua, only through the C API. This guarantees the integrity of data owned by the host program.

The type *thread* represents independent threads of execution and it is used to implement coroutines (see §2.6). Do not confuse Lua threads with operating-system threads. Lua supports coroutines on all systems, even those that do not support threads.

The type *table* implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any Lua value except **nil** and NaN (*Not a Number*, a special numeric value used to represent undefined or unrepresentable results, such as `0/0`). Tables can be *heterogeneous*; that is, they can contain values of all types (except **nil**). Any key with value **nil** is not considered part of the table. Conversely, any key that is not part of a table has an associated value **nil**.

Tables are the sole data structuring mechanism in Lua; they can be used to represent ordinary arrays, sequences, symbol tables, sets, records, graphs, trees, etc. To represent records, Lua uses the field name as an index. The language supports this representation by providing `a.name` as syntactic sugar for `a["name"]`. There are several convenient ways to create tables in Lua (see §3.4.8).

We use the term *sequence* to denote a table where the set of all positive numeric keys is equal to $1..n$ for some integer n, which is called the length of the sequence (see §3.4.6).

Like indices, the values of table fields can be of any type. In particular, because functions are first-class values, table fields can contain functions. Thus tables can also carry *methods* (see §3.4.10).

The indexing of tables follows the definition of raw equality in the language. The expressions `a[i]` and `a[j]` denote the same table element if and only if `i` and `j` are raw equal (that is, equal without metamethods).

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

The library function `type` returns a string describing the type of a given value (see §6.1).

2.2 · Environments and the Global Environment

As will be discussed in §3.2 and §3.3.3, any reference to a global name `var` is syntactically translated to `_ENV.var`. Moreover, every chunk is compiled in the scope of an external local variable called `_ENV` (see §3.3.2), so `_ENV` itself is never a global name in a chunk.

Despite the existence of this external `_ENV` variable and the translation of global names, `_ENV` is a completely regular name. In particular, you can define new variables and parameters with that name. Each reference to a global name uses the `_ENV` that is visible at that point in the program, following the usual visibility rules of Lua (see §3.5).

Any table used as the value of `_ENV` is called an *environment*.

Lua keeps a distinguished environment called the *global environment*. This value is kept at a special index in the C registry (see §4.5). In Lua, the variable `_G` is initialized with this same value.

When Lua compiles a chunk, it initializes the value of its `_ENV` upvalue with the global environment (see `load`). Therefore, by default, global variables in Lua code refer to entries in the global environment. Moreover, all standard libraries are loaded in the global environment and several functions there operate on that environment. You can use `load` (or `loadfile`) to load a chunk with a different environment. (In C, you have to load the chunk and then change the value of its first upvalue.)

If you change the global environment in the registry (through C code or the debug library), all chunks loaded after the change will get the new environment. Previously loaded chunks are not affected, however, as each has its own reference to the environment in its `_ENV` variable. Moreover, the variable `_G` (which is stored in the original global environment) is never updated by Lua.

2.3 · Error Handling

Because Lua is an embedded extension language, all Lua actions start from C code in the host program calling a function from the Lua library (see `lua_pcall`). Whenever an error occurs during the compilation or execution of a Lua chunk, control returns to the host, which can take appropriate measures (such as printing an error message).

Lua code can explicitly generate an error by calling the `error` function. If you need to catch errors in Lua, you can use `pcall` or `xpcall` to call a given function in *protected mode*.

Whenever there is an error, an *error object* (also called an *error message*) is propagated with information about the error. Lua itself only generates errors where the error object is a string, but programs may generate errors with any value for the error object.

When you use `xpcall` or `lua_pcall`, you may give a *message handler* to be called in case of errors. This function is called with the original error message and returns a new error message. It is called before the error unwinds the stack, so that it can gather more information about the error, for instance by inspecting the stack and creating a stack traceback. This message handler is still protected by the protected call; so, an error inside the message handler will call the message handler again. If this loop goes on, Lua breaks it and returns an appropriate message.

2.4 · Metatables and Metamethods

Every value in Lua can have a *metatable*. This *metatable* is an ordinary Lua table that defines the behavior of the original value under certain special operations. You can change several aspects of the behavior of operations over a value by setting specific fields in its metatable. For instance, when a non-numeric value is the operand of an addition, Lua checks for a function in the field `__add` of the value's metatable. If it finds one, Lua calls this function to perform the addition.

The keys in a metatable are derived from the *event* names; the corresponding values are called *metamethods*. In the previous example, the event is `"add"` and the metamethod is the function that performs the addition.

You can query the metatable of any value using the `getmetatable` function.

You can replace the metatable of tables using the `pdf-setmetatablesetmetatable` function. You cannot change the metatable of other types from Lua (except by using the debug library); you must use the C API for that.

Tables and full userdata have individual metatables (although multiple tables and userdata can share their metatables). Values of all other types share one single metatable per type; that is, there is one single metatable for all numbers, one for all strings, etc. By default, a value has no metatable, but the string library sets a metatable for the string type (see §6.4).

A metatable controls how an object behaves in arithmetic operations, order comparisons, concatenation, length operation, and indexing. A metatable also can define a function to be called when a userdata or a table is garbage collected. When Lua performs one of these operations over a value, it checks whether this value has a metatable with the corresponding event. If so, the value associated with that key (the metamethod) controls how Lua will perform the operation.

Metatables control the operations listed next. Each operation is identified by its corresponding name. 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"`.

The semantics of these operations is better explained by a Lua function describing how the interpreter executes the operation. The code shown here in Lua is only illustrative; the real behavior is hard coded in the interpreter and it is much more efficient than this simulation. All functions used in these descriptions (`rawget`, `tonumber`, etc.) are described in §6.1. In particular, to retrieve the metamethod of a given object, we use the expression

```
metatable(obj)[event]
```

This should be read as

```
rawget(getmetatable(obj) or {}, event)
```

This means that the access to a metamethod does not invoke other metamethods, and access to objects with no metatables does not fail (it simply results in **nil**).

For the unary `-` and `#` operators, the metamethod is called with a dummy second argument. This extra argument is only to simplify Lua's internals; it may be removed in future versions and therefore it is not present in the following code. (For most uses this extra argument is irrelevant.)

- `"add"`: the `+` operation.
  The function `getbinhandler` below defines how Lua chooses a handler for a binary operation. First, Lua tries the first operand. If its type does not define a handler for the operation, then Lua tries the second operand.

```
function getbinhandler (op1, op2, event)
  return metatable(op1)[event] or
         metatable(op2)[event]
end
```

  By using this function, the behavior of `op1+op2` is

```
function add_event (op1, op2)
  local o1,o2 = tonumber(op1),tonumber(op2)
  if o1 and o2 then  -- both are numeric?
    return o1 + o2   -- '+' here is 'add'
  else  -- at least one is not numeric
    local h = getbinhandler(op1,op2,"__add")
    if h then
      -- call handler with both operands
      return (h(op1, op2))
    else  -- no handler available
      error(···)
    end
  end
end
```

- `"sub"`: the `-` operation. Behavior similar to the "add" operation.
- `"mul"`: the `*` operation. Behavior similar to the "add" operation.
- `"div"`: the `/` operation. Behavior similar to the "add" operation.
- `"mod"`: the `%` operation. Behavior similar to the "add" operation, with the operation `o1 - floor(o1/o2)*o2` as the primitive operation.
- `"pow"`: the `^` (exponentiation) operation. Behavior similar to the "add" operation, with the function `pow` (from the C math library) as the primitive operation.
- `"unm"`: the unary `-` operation.

```
function unm_event (op)
  local o = tonumber(op)
  if o then  -- operand is numeric?
    return -o  -- '-' here is 'unm'
  else  -- the operand is not numeric.
    -- Try to get a handler
    local h = metatable(op).__unm
    if h then
      -- call the handler
      return (h(op))
    else  -- no handler available
      error(···)
    end
  end
end
```

- `"concat"`: the `..` (concatenation) operation.

```
function concat_event (op1, op2)
  if (type(op1) == "string" or
      type(op1) == "number") and
     (type(op2) == "string" or
      type(op2) == "number") then
    return op1 .. op2  -- primitive
  else
    local h = getbinhandler(op1,op2,
                            "__concat")
    if h then
      return (h(op1, op2))
    else
      error(···)
    end
  end
end
```

- `"len"`: the `#` operation.

```
function len_event (op)
  if type(op) == "string" then
    return strlen(op)      -- primitive
  else
    local h = metatable(op).__len
    if h then
      return (h(op))       -- call handler
    elseif type(op) == "table" then
      return #op              -- primitive
    else  -- no handler available: error
      error(···)
    end
  end
end
```

  See §3.4.6 for a description of the length of a table.

- `"eq"`: the `==` operation. The `getequalhandler` function defines how Lua chooses a metamethod for equality. A metamethod is selected only when both values being compared have the same type and the same metamethod for the selected operation, and the values are either tables or full userdata.

```
function getequalhandler (op1, op2)
  if type(op1) ~= type(op2) or
     (type(op1) ~= "table" and
      type(op1) ~= "userdata") then
    return nil     -- different values
  end
  local mm1 = metatable(op1).__eq
  local mm2 = metatable(op2).__eq
  if mm1 == mm2 then return mm1
                else return nil end
end
```

  The '`eq`' event is defined as follows:

```
function eq_event (op1, op2)
  if op1 == op2 then   -- primitive equal?
    return true   -- values are equal
  end
  -- try metamethod
  local h = getequalhandler(op1, op2)
  if h then
    return not not h(op1, op2)
```

```
  else
    return false
  end
end
```

Note that the result is always a boolean.

- `"lt"`: the `<` operation.

```
function lt_event (op1, op2)
  if type(op1) == "number" and
     type(op2) == "number" then
    return op1 < op2   -- numeric
  elseif type(op1) == "string" and
         type(op2) == "string" then
    return op1 < op2   -- lexicographic
  else
    local h = getbinhandler(op1,op2,"__lt")
    if h then
      return not not h(op1, op2)
    else
      error(···)
    end
  end
end
```

Note that the result is always a boolean.

- `"le"`: the `<=` operation.

```
function le_event (op1, op2)
  if type(op1) == "number" and
     type(op2) == "number" then
    return op1 <= op2   -- numeric
  elseif type(op1) == "string" and
         type(op2) == "string" then
    return op1 <= op2   -- lexicographic
  else
    local h = getbinhandler(op1,op2,"__le")
    if h then
      return not not h(op1, op2)
    else
      h = getbinhandler(op1,op2,"__lt")
      if h then
        return not h(op2, op1)
      else
        error(···)
      end
    end
  end
end
```

Note that, in the absence of a `"le"` metamethod, Lua tries `"lt"`, assuming that `a<=b` is equivalent to `not (b<a)`.
As with the other comparison operators, the result is always a boolean.

- `"index"`: The indexing access `table[key]`. Note that the metamethod is tried only when `key` is not present in `table`. (When `table` is not a table, no key is ever present, so the metamethod is always tried.)

```
function gettable_event (table, key)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    -- if key is present, return raw value
    if v ~= nil then return v end
    h = metatable(table).__index
    if h == nil then return nil end
  else
    h = metatable(table).__index
    if h == nil then
      error(···)
    end
  end
  if type(h) == "function" then
    return (h(table, key))  -- call handler
  else return h[key]        -- or repeat
  end
end
```

- `"newindex"`: The indexing assignment `table[key] = value`. Note that the metamethod is tried only when `key` is not present in `table`.

```
function settable_event (table, key, value)
  local h
  if type(table) == "table" then
    local v = rawget(table, key)
    -- if key is present, do raw assignment
    if v ~= nil then
      rawset(table, key, value); return end
    h = metatable(table).__newindex
    if h == nil then
      rawset(table, key, value); return end
  else
    h = metatable(table).__newindex
    if h == nil then
      error(···)
    end
  end
  if type(h) == "function" then
    h(table, key,value)  -- call handler
  else h[key] = value    -- or repeat
  end
end
```

- `"call"`: called when Lua calls a value.

```
function function_event (func, ...)
  if type(func) == "function" then
    return func(...)   -- primitive call
  else
    local h = metatable(func).__call
    if h then
      return h(func, ...)
    else
      error(···)
    end
  end
end
```

2.5 · Garbage Collection

Lua performs automatic memory management. This means that you have to worry neither about allocating memory for new objects nor about freeing it when the objects are no longer needed. Lua manages memory automatically by running a *garbage collector* to collect all *dead objects* (that is, objects that are no longer accessible from Lua). All memory used by Lua is subject to automatic management: strings, tables, userdata, functions, threads, internal structures, etc.

Lua implements an incremental mark-and-sweep collector. It uses two numbers to control its garbage-collection cycles: the *garbage-collector pause* and the *garbage-collector step multiplier*. Both use percentage points as units (e. g., a value of 100 means an internal value of 1).

The garbage-collector pause controls how long the collector waits before starting a new cycle. Larger values make the collector less aggressive. Values smaller than 100 mean the collector will not wait to start a new cycle. A value of 200 means that the collector waits for the total memory in use to double before starting a new cycle.

The garbage-collector step multiplier controls the relative speed of the collector relative to memory allocation. Larger values make the collector more aggressive but also increase the size of each incremental step. Values smaller than 100 make the collector too slow and can result in the collector never finishing a cycle. The default is 200, which means that the collector runs at 'twice' the speed of memory allocation.

If you set the step multiplier to a very large number (larger than 10% of the maximum number of bytes that the program may use), the collector behaves like a stop-the-world collector. If you then set the pause to 200, the collector behaves as in old Lua versions, doing a complete collection every time Lua doubles its memory usage.

You can change these numbers by calling `lua_gc` in C or `collectgarbage` in Lua. You can also use these functions to control the collector directly (e.g., stop and restart it).

As an experimental feature in Lua 5.2, you can change the collector's operation mode from incremental to *generational*. A *generational collector* assumes that most objects die young, and therefore it traverses only young (recently created) objects. This behavior can reduce the time used by the collector, but also increases memory usage (as old dead objects may accumulate). To mitigate this second problem, from time to time the generational collector performs a full collection. Remember that this is an experimental feature; you are welcome to try it, but check your gains.

2.5.1 · Garbage-Collection Metamethods

You can set garbage-collector metamethods for tables and, using the C API, for full userdata (see §2.4). These metamethods are also called *finalizers*. Finalizers allow you to coordinate Lua's garbage collection with external resource management (such as closing files, network or database connections, or freeing your own memory).

For an object (table or userdata) to be finalized when collected, you must *mark* it for finalization.

You mark an object for finalization when you set its metatable and the metatable has a field indexed by the string `"__gc"`. Note that if you set a metatable without a `__gc` field and later create that field in the metatable, the object will not be marked for finalization. However, after an object is marked, you can freely change the `__gc` field of its metatable.

When a marked object becomes garbage, it is not collected immediately by the garbage collector. Instead, Lua puts it in a list. After the collection, Lua does the equivalent of the following function for each object in that list:

```
function gc_event (obj)
  local h = metatable(obj).__gc
  if type(h) == "function" then
    h(obj)
  end
end
```

At the end of each garbage-collection cycle, the finalizers for objects are called in the reverse order that they were marked for collection, among those collected in that cycle; that is, the first finalizer to be called is the one associated with the object marked last in the program. The execution of each finalizer may occur at any point during the execution of the regular code.

Because the object being collected must still be used by the finalizer, it (and other objects accessible only through it) must be *resurrected* by Lua. Usually, this resurrection is transient, and the object memory is freed in the next garbage-collection cycle. However, if the finalizer stores the object in some global place (e.g., a global variable), then there is a permanent resurrection. In any case, the object memory is freed only when it becomes completely inaccessible; its finalizer will never be called twice.

When you close a state (see `lua_close`), Lua calls the finalizers of all objects marked for collection, following the reverse order that they were marked. If any finalizer marks new objects for collection during that phase, these new objects will not be finalized.

2.5.2 · Weak Tables

A *weak table* is a table whose elements are *weak references*. A weak reference is ignored by the garbage collector. In other words, if the only references to an object are weak references, then the garbage collector will collect that object.

A weak table can have weak keys, weak values, or both. A table with weak keys allows the collection of its keys, but prevents the collection of its values. A table with both weak keys and weak values allows the collection of both keys and values. In any case, if either the key or the value is collected, the whole pair is removed from the table. The weakness of a table is controlled by the `__mode` field of its metatable. If the `__mode` field is a string containing the character `'k'`, the keys in the table are weak. If `__mode` contains `'v'`, the values in the table are weak.

A table with weak keys and strong values is also called an *ephemeron table*. In an ephemeron table, a value is considered reachable only if its key is reachable. In particular, if the only reference to a key comes through its value, the pair is removed.

Any change in the weakness of a table may take effect only at the next collect cycle. In particular, if you change the weakness to a stronger mode, Lua may still collect some items from that table before the change takes effect.

Only objects that have an explicit construction are removed from weak tables. Values, such as numbers and light C functions, are not subject to garbage collection, and therefore are not removed from weak tables (unless

its associated value is collected). Although strings are subject to garbage collection, they do not have an explicit construction, and therefore are not removed from weak tables.

Resurrected objects (that is, objects being finalized and objects accessible only through objects being finalized) have a special behavior in weak tables. They are removed from weak values before running their finalizers, but are removed from weak keys only in the next collection after running their finalizers, when such objects are actually freed. This behavior allows the finalizer to access properties associated with the object through weak tables.

If a weak table is among the resurrected objects in a collection cycle, it may not be properly cleared until the next cycle.

2.6 · Coroutines

Lua supports coroutines, also called *collaborative multithreading*. A coroutine in Lua represents an independent thread of execution. Unlike threads in multithread systems, however, a coroutine only suspends its execution by explicitly calling a yield function.

You create a coroutine by calling `coroutine.create`. Its sole argument is a function that is the main function of the coroutine. The `create` function only creates a new coroutine and returns a handle to it (an object of type *thread*); it does not start the coroutine.

You execute a coroutine by calling `coroutine.resume`. When you first call `coroutine.resume`, passing as its first argument a thread returned by `coroutine.create`, the coroutine starts its execution, at the first line of its main function. Extra arguments passed to `coroutine.resume` are passed on to the coroutine main function. After the coroutine starts running, it runs until it terminates or *yields*.

A coroutine can terminate its execution in two ways: normally, when its main function returns (explicitly or implicitly, after the last instruction); and abnormally, if there is an unprotected error. In the first case, `coroutine.resume` returns `true`, plus any values returned by the coroutine main function. In case of errors, `coroutine.resume` returns `false` plus an error message.

A coroutine yields by calling `coroutine.yield`. When a coroutine yields, the corresponding `coroutine.resume` returns immediately, even if the yield happens inside nested function calls (that is, not in the main function, but in a function directly or indirectly called by the main function). In the case of a yield, `coroutine.resume` also returns `true`, plus any values passed to `coroutine.yield`. The next time you resume the same coroutine, it continues its execution from the point where it yielded, with the call to `coroutine.yield` returning any extra arguments passed to `coroutine.resume`.

Like `coroutine.create`, the `coroutine.wrap` function also creates a coroutine, but instead of returning the coroutine itself, it returns a function that, when called, resumes the coroutine. Any arguments passed to this function go as extra arguments to `coroutine.resume`. `coroutine.wrap` returns all the values returned by `coroutine.resume`, except the first one (the boolean error code). Unlike `coroutine.resume`, `coroutine.wrap` does not catch errors; any error is propagated to the caller.

As an example of how coroutines work, consider the following code:

```
function foo (a)
  print("foo", a)
  return coroutine.yield(2*a)
end

co = coroutine.create(function (a,b)
      print("co-body", a, b)
      local r = foo(a+1)
      print("co-body", r)
      local r,s = coroutine.yield(a+b,a-b)
      print("co-body",r,s)
      return b,"end"
end)

print("main",coroutine.resume(co,1,10))
print("main",coroutine.resume(co,"r"))
print("main",coroutine.resume(co,"x","y"))
print("main",coroutine.resume(co,"x","y"))
```

When you run it, it produces the following output:

```
co-body 1     10
foo     2
main    true  4
co-body r
main    true  11  -9
co-body x     y
main    true  10  end
main    false cannot resume dead coroutine
```

You can also create and manipulate coroutines through the C API: see functions `lua_newthread`, `lua_resume`, and `lua_yield`.

3 · The Language

This section describes the lexis, the syntax, and the semantics of Lua. In other words, this section describes which tokens are valid, how they can be combined, and what their combinations mean.

Language constructs will be explained using the usual extended BNF notation, in which `{`*a*`}` means 0 or more *a*'s, and `[`*a*`]` means an optional *a*. The complete syntax of Lua can be found in §9 at the end of this manual.

3.1 · Lexical Conventions

Lua is a free-form language. It ignores spaces (including new lines) and comments between lexical elements (tokens), except as delimiters between names and keywords.

Names (also called *identifiers*) in Lua can be any string of letters, digits, and underscores, not beginning with a digit. Identifiers are used to name variables, table fields, and labels.

The following *keywords* are reserved and cannot be used as names:

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

Lua is a case-sensitive language: `and` is a reserved word, but `And` and `AND` are two different, valid names. As a convention, names starting with an underscore followed by uppercase letters (such as `_VERSION`) are reserved for variables used by Lua.

The following strings denote other tokens:

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

Literal strings can be delimited by matching single or double quotes, and can contain the following C-like escape sequences: `\a` (bell), `\b` (backspace), `\f` (form feed), `\n` (newline), `\r` (carriage return), `\t` (horizontal tab), `\v` (vertical tab), `\\` (backslash), `\"` (quotation mark [double quote]), and `\'` (apostrophe [single quote]). A backslash followed by a real newline results in a newline in the string. The escape sequence `\z` skips the following span of white-space characters, including line breaks; it is particularly useful to break and indent a long literal string into multiple lines without adding the newlines and spaces into the string contents.

A byte in a literal string can also be specified by its numerical value. This can be done with the escape sequence `\x`XX, where XX is a sequence of exactly two hexadecimal digits, or with the escape sequence \ddd, where ddd is a sequence of up to three decimal digits. (Note that if a decimal escape is to be followed by a digit, it must be expressed using exactly three digits.) Strings in Lua can contain any 8-bit value, including embedded zeros, which can be specified as '`\0`'.

Literal strings can also be defined using a long format enclosed by *long brackets*. We define an *opening long bracket of level n* as an opening square bracket followed by n equal signs followed by another opening square bracket. So, an opening long bracket of level 0 is written as `[[`, an opening long bracket of level 1 is written as `[=[`, and so on. A *closing long bracket* is defined similarly; for instance, a closing long bracket of level 4 is written as `]====]`. A *long literal* starts with an opening long bracket of any level and ends at the first closing long bracket of the same level. It can contain any text except a closing bracket of the proper level. Literals in this bracketed form can run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. Any kind of end-of-line sequence (carriage return, newline, carriage return followed by newline, or newline followed by carriage return) is converted to a simple newline.

When parsing a from a string source, any byte in a literal string not explicitly affected by the previous rules represents itself. However, Lua opens files for parsing in text mode, and the system file functions may have problems with some control characters. So, it is safer to represent non-text data as a quoted literal with explicit escape sequences for non-text characters.

For convenience, when the opening long bracket is immediately followed by a newline, the newline is not included in the string. As an example, in a system using ASCII (in which `'a'` is coded as 97, newline is coded as 10, and `'1'` is coded as 49), the five literal strings below denote the same string:

```
a = 'alo\n123"'
a = "alo\n123\""
a = '\97lo\10\04923"'
a = [[alo
123"]]
a = [==[
alo
123"]==]
```

A *numerical constant* can be written with an optional fractional part and an optional decimal exponent, marked by a letter `e` or `E`. Lua also accepts hexadecimal constants, which start with `0x` or `0X`. Hexadecimal constants also accept an optional fractional part plus an optional binary exponent, marked by a letter `p` or `P`. Examples of valid numerical constants are

```
3     3.0     3.1416    314.16e-2    0.31416E1
0xff  0x0.1E  0xA23p-4  0X1.921FB54442D18P+1
```

A *comment* starts with a double hyphen (`--`) anywhere outside a string. If the text immediately after `--` is not an opening long bracket, the comment is a *short comment*, which runs until the end of the line. Otherwise, it is a *long comment*, which runs until the corresponding closing long bracket. Long comments are frequently used to disable code temporarily.

3.2 · Variables

Variables are places that store values. There are three kinds of variables in Lua: global variables, local variables, and table fields.

A single name can denote a global variable or a local variable (or a function's formal parameter, which is a particular kind of local variable):

```
var ::= Name
```

Names denote identifiers, as defined in §3.1.

Any variable name is assumed to be global unless explicitly declared as a local (see §3.3.7). Local variables are *lexically scoped*: local variables can be freely accessed by functions defined inside their scope (see §3.5).

Before the first assignment to a variable, its value is `nil`.

Square brackets are used to index a table:

```
var ::= prefixexp '[' exp ']'
```

The meaning of accesses to table fields can be changed via metatables. An access to an indexed variable `t[i]` is equivalent to a call `gettable_event(t,i)`. (See §2.4 for a complete description of the `gettable_event` function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)

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

```
var ::= prefixexp '.' Name
```

An access to a global variable `x` is equivalent to `_ENV.x`. Due to the way that chunks are compiled, `_ENV` is never a global name (see §2.2).

3.3 · Statements

Lua supports an almost conventional set of statements, similar to those in Pascal or C. This set includes assignments, control structures, function calls, and variable declarations.

3.3.1 · Blocks

A block is a list of statements, which are executed sequentially:

```
block ::= {stat}
```

Lua has *empty statements* that allow you to separate statements with semicolons, start a block with a semicolon or write two semicolons in sequence:

```
stat ::= ';'
```

A block can be explicitly delimited to produce a single statement:

```
stat ::= do block end
```

Explicit blocks are useful to control the scope of variable declarations. Explicit blocks are also sometimes used to add a `return` statement in the middle of another block (see §3.3.4).

3.3.2 · Chunks

The unit of execution of Lua is called a *chunk*. Syntactically, a chunk is simply a block:

```
chunk ::= block
```

Lua handles a chunk as the body of an anonymous function with a variable number of arguments (see §3.4.10). As such, chunks can define local variables, receive arguments, and return values. Moreover, such anonymous function is compiled as in the scope of an external local variable called `_ENV` (see §2.2). The resulting function always has `_ENV` as its only upvalue, even if it does not use that variable.

A chunk can be stored in a file or in a string inside the host program. To execute a chunk, Lua first precompiles the chunk into instructions for a virtual machine, and then it executes the compiled code with an interpreter for the virtual machine.

Chunks can also be precompiled into binary form; see program `luac` for details. Programs in source and compiled forms are interchangeable; Lua automatically detects the file type and acts accordingly.

3.3.3 · Assignment

Lua allows multiple assignments. Therefore, the syntax for assignment defines a list of variables on the left side and a list of expressions on the right side. The elements in both lists are separated by commas:

```
stat ::= varlist '=' explist
varlist ::= var {',' var}
explist ::= exp {',' exp}
```

Expressions are discussed in §3.4.

Before the assignment, the list of values is *adjusted* to the length of the list of variables. If there are more values than needed, the excess values are thrown away. If there are fewer values than needed, the list is extended with as many `nil`s as needed. If the list of expressions ends with a function call, then all values returned by that call enter the list of values, before the adjustment (except when the call is enclosed in parentheses; see §3.4).

The assignment statement first evaluates all its expressions and only then are the assignments performed. Thus the code

```
i = 3
i, a[i] = i+1, 20
```

sets `a[3]` to 20, without affecting `a[4]` because the `i` in `a[i]` is evaluated (to 3) before it is assigned 4. Similarly, the line

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

exchanges the values of `x` and `y`, and

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

cyclically permutes the values of `x`, `y`, and `z`.

The meaning of assignments to global variables and table fields can be changed via metatables. An assignment to an indexed variable `t[i] = val` is equivalent to `settable_event(t,i,val)`. (See §2.4 for a complete description of the `settable_event` function. This function is not defined or callable in Lua. We use it here only for explanatory purposes.)

An assignment to a global variable `x = val` is equivalent to the assignment `_ENV.x = val` (see §2.2).

3.3.4 · Control Structures

The control structures `if`, `while`, and `repeat` have the usual meaning and familiar syntax:

```
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block
         {elseif exp then block}
         [else block] end
```

Lua also has a `for` statement, in two flavors (see §3.3.5).

The condition expression of a control structure can return any value. Both `false` and `nil` are considered false. All values different from `nil` and `false` are considered true (in particular, the number 0 and the empty string are also true).

In the `repeat`—`until` loop, the inner block does not end at the `until` keyword, but only after the condition. So, the condition can refer to local variables declared inside the loop block.

The `goto` statement transfers the program control to a label. For syntactical reasons, labels in Lua are considered statements too:

```
stat ::= goto Name
stat ::= label
label ::= ‘::’ Name ‘::’
```

A label is visible in the entire block where it is defined, except inside nested blocks where a label with the same name is defined and inside nested functions. A goto may jump to any visible label as long as it does not enter into the scope of a local variable.

Labels and empty statements are called *void statements*, as they perform no actions.

The `break` statement terminates the execution of a `while`, `repeat`, or `for` loop, skipping to the next statement after the loop:

```
stat ::= break
```

A `break` ends the innermost enclosing loop.

The `return` statement is used to return values from a function or a chunk (which is a function in disguise). Functions can return more than one value, so the syntax for the `return` statement is

```
stat ::= return [explist] [‘;’]
```

The `return` statement can only be written as the last statement of a block. If it is really necessary to `return` in the middle of a block, then an explicit inner block can be used, as in the idiom `do return end`, because now `return` is the last statement in its (inner) block.

3.3.5 · For Statement

The `for` statement has two forms: one numeric and one generic.

The numeric `for` loop repeats a block of code while a control variable runs through an arithmetic progression. It has the following syntax:

```
stat ::= for Name ‘=’ exp ‘,’ exp [‘,’ exp]
         do block end
```

The `block` is repeated for `name` starting at the value of the first `exp`, until it passes the second `exp` by steps of the third `exp`. More precisely, a `for` statement like

```
for v = e1, e2, e3 do block end
```

is equivalent to the code:

```
do
  local var,limit,step =
        tonumber(e1),tonumber(e2),tonumber(e3)
  if not (var and limit and step)
    then error() end
  while (step > 0 and var <= limit) or
        (step <= 0 and var >= limit) do
    local v = var
    block
    var = var + step
  end
end
```

Note the following:

- All three control expressions are evaluated only once, before the loop starts. They must all result in numbers.
- `var`, `limit`, and `step` are invisible variables. The names shown here are for explanatory purposes only.
- If the third expression (the step) is absent, then a step of 1 is used.
- You can use `break` to exit a `for` loop.
- The loop variable `v` is local to the loop; you cannot use its value after the `for` ends or is broken. If you need this value, assign it to another variable before breaking or exiting the loop.

The generic `for` statement works over functions, called *iterators*. On each iteration, the iterator function is called to produce a new value, stopping when this new value is `nil`. The generic `for` loop has the following syntax:

```
stat ::= for namelist in explist do block end
namelist ::= Name {‘,’ Name}
```

A `for` statement like

```
for var_1, ···, var_n in explist do block end
```

is equivalent to the code:

```
do
  local f, s, var = explist
  while true do
    local var_1, ···, var_n = f(s, var)
    if var_1 == nil then break end
    var = var_1
    block
  end
end
```

Note the following:

- `explist` is evaluated only once. Its results are an iterator function, a state, and an initial value for the first iterator variable.
- `f`, `s`, and `var` are invisible variables. The names are here for explanatory purposes only.
- You can use `break` to exit a `for` loop.
- The loop variables `var_i` are local to the loop; you cannot use their values after the `for` ends. If you need these values, then assign them to other variables before breaking or exiting the loop.

3.3.6 · Function Calls as Statements

To allow possible side-effects, function calls can be executed as statements:

```
stat ::= functioncall
```

In this case, all returned values are thrown away. Function calls are explained in §3.4.9.

3.3.7 · Local Declarations

Local variables can be declared anywhere inside a block. The declaration can include an initial assignment:

```
stat ::= local namelist [‘=’ explist]
```

If present, an initial assignment has the same semantics of a multiple assignment (see §3.3.3). Otherwise, all variables are initialized with `nil`.

A chunk is also a block (see §3.3.2), and so local variables can be declared in a chunk outside any explicit block.

The visibility rules for local variables are explained in §3.5.

3.4 · Expressions

The basic expressions in Lua are the following:

```
exp ::= prefixexp
exp ::= nil | false | true
exp ::= Number
exp ::= String
exp ::= functiondef
exp ::= tableconstructor
exp ::= ‘...’
exp ::= exp binop exp
exp ::= unop exp
prefixexp ::= var | functioncall | ‘(’ exp ‘)’
```

Numbers and literal strings are explained in §3.1; variables are explained in §3.2; function definitions are explained in §3.4.10; function calls are explained in §3.4.9; table constructors are explained in §3.4.8. Vararg expressions, denoted by three dots (’`...`’), can only be used when directly inside a vararg function; they are explained in §3.4.10.

Binary operators comprise arithmetic operators (see §3.4.1), relational operators (see §3.4.3), logical operators (see §3.4.4), and the concatenation operator (see §3.4.5). Unary operators comprise the unary minus (see §3.4.1), the unary `not` (see §3.4.4), and the unary *length operator* (see §3.4.6).

Both function calls and vararg expressions can result in multiple values. If a function call is used as a statement (see §3.3.6), then its return list is adjusted to zero elements, thus discarding all returned values. If an expression is used as the last (or the only) element of a list of expressions, then no adjustment is made (unless the expression is enclosed in parentheses). In all other contexts, Lua adjusts the result list to one element, discarding all values except the first one.

Here are some examples:

```
f()             -- adjusted to 0 results
g(f(),x)        -- f() is adjusted to 1 result
g(x,f())        -- g gets x plus all results
                -- from f()
a,b,c = f(),x   -- f() is adjusted to 1 result
                -- (c gets nil)
a,b = ...       -- a gets the 1st vararg
                -- parameter, b gets the 2nd
                -- (both a and b can get nil
                -- if there is no corresponding
                -- vararg parameter)
a,b,c = x,f()   -- f() is adjusted to 2 results
a,b,c = f()     -- f() is adjusted to 3 results
return f()      -- returns all results from f()
return ...      -- returns all received vararg
                -- parameters
return x,y,f()  -- returns x, y, and all
                -- results from f()
{f()}           -- creates a list with all
                -- results from f()
{...}           -- creates a list with all
                -- vararg parameters
{f(),nil}       -- f() is adjusted to 1 result
```

Any expression enclosed in parentheses always results in only one value. Thus, `(f(x,y,z))` is always a single value, even if `f` returns several values. (The value of `(f(x,y,z))` is the first value returned by `f` or `nil` if `f` does not return any values.)

3.4.1 · Arithmetic Operators

Lua supports the usual arithmetic operators: the binary `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulo), and `^` (exponentiation); and unary `-` (mathematical negation). If the operands are numbers, or strings that can be converted to numbers (see §3.4.2), then all operations have the usual meaning. Exponentiation works for any exponent. For instance, `x^(-0.5)` computes the inverse of the square root of `x`. Modulo is defined as

```
a % b == a - math.floor(a/b)*b
```

That is, it is the remainder of a division that rounds the quotient towards minus infinity.

3.4.2 · Coercion

Lua provides automatic conversion between string and number values at run time. Any arithmetic operation applied to a string tries to convert this string to a number, following the rules of the Lua lexer. (The string may have leading and trailing spaces and a sign.) Conversely, whenever a number is used where a string is expected, the number is converted to a string, in a reasonable format. For complete control over how numbers are converted to strings, use the `format` function from the string library (see `string.format`).

3.4.3 · Relational Operators

The relational operators in Lua are

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

These operators always result in `false` or `true`.

Equality (`==`) first compares the type of its operands. If the types are different, then the result is `false`. Otherwise, the values of the operands are compared. Numbers and strings are compared in the usual way. Tables, userdata, and threads are compared by reference: two objects are considered equal only if they are the same object.

Every time you create a new object (a table, userdata, or thread), this new object is different from any previously existing object. Closures with the same reference are always equal. Closures with any detectable difference (different behavior, different definition) are always different.

You can change the way that Lua compares tables and userdata by using the "eq" metamethod (see §2.4).

The conversion rules of §3.4.2 do not apply to equality comparisons. Thus, `"0"==0` evaluates to `false`, and `t[0]` and `t["0"]` denote different entries in a table.

The `~=` operator is the negation of equality (`==`).

The order operators work as follows. If both arguments are numbers, then they are compared as such. Otherwise, if both arguments are strings, then their values are compared according to the current locale. Otherwise, Lua tries to call the `"lt"` or the `"le"` metamethod (see §2.4). A comparison `a>b` is translated to `b<a` and `a>=b` is translated to `b<=a`.

3.4.4 · Logical Operators

The logical operators in Lua are `and`, `or`, and `not`. Like the control structures (see §3.3.4), all logical operators consider both `false` and `nil` as false and anything else as true.

The negation operator `not` always returns `false` or `true`. The conjunction operator `and` returns its first argument if this value is `false` or `nil`; otherwise, `and` returns its second argument. The disjunction operator `or` returns its first argument if this value is different from `nil` and `false`; otherwise, `or` returns its second argument. Both `and` and `or` use short-cut evaluation; that is, the second operand is evaluated only if necessary. Here are some examples:

```
10 or 20            --> 10
10 or error()       --> 10
nil or "a"          --> "a"
nil and 10          --> nil
false and error()   --> false
false and nil       --> false
false or nil        --> nil
10 and 20           --> 20
```

(In this manual, `-->` indicates the result of the preceding expression.)

3.4.5 · Concatenation

The string concatenation operator in Lua is denoted by two dots (`..`). If both operands are strings or numbers, then they are converted to strings according to the rules mentioned in §3.4.2. Otherwise, the `__concat` metamethod is called (see §2.4).

3.4.6 · The Length Operator

The length operator is denoted by the unary prefix operator `#`. The length of a string is its number of bytes (that is, the usual meaning of string length when each character is one byte).

A program can modify the behavior of the length operator for any value but strings through the `__len` metamethod (see §2.4).

Unless a `__len` metamethod is given, the length of a table `t` is only defined if the table is a *sequence*, that is, the set of its positive numeric keys is equal to $1..n$ for some integer n. In that case, n is its length. Note that a table like

```
{10, 20, nil, 40}
```

is not a sequence, because it has the key `4` but does not have the key `3`. (So, there is no n such that the set $1..n$ is equal to the set of positive numeric keys of that table.) Note, however, that non-numeric keys do not interfere with whether a table is a sequence.

3.4.7 · Precedence

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

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

As usual, you can use parentheses to change the precedences of an expression. The concatenation (`..`) and exponentiation (`^`) operators are right associative. All other binary operators are left associative.

3.4.8 · Table Constructors

Table constructors are expressions that create tables. Every time a constructor is evaluated, a new table is created. A constructor can be used to create an empty table or to create a table and initialize some of its fields. The general syntax for constructors is

```
tableconstructor ::= '{' [fieldlist] '}'
fieldlist ::= field {fieldsep field}
              [fieldsep]
field ::= '[' exp ']' '=' exp | Name '=' exp
          | exp
fieldsep ::= ',' | ';'
```

Each field of the form `[exp1] = exp2` adds to the new table an entry with key `exp1` and value `exp2`. A field of the form `name = exp` is equivalent to `["name"] = exp`. Finally, fields of the form `exp` are equivalent to `[i] = exp`, where `i` are consecutive numerical integers, starting with 1. Fields in the other formats do not affect this counting. For example,

```
a = { [f(1)] = g; "x", "y"; x = 1, f(x),
      [30] = 23; 45 }
```

is equivalent to

```
do
  local t = {}
  t[f(1)] = g
  t[1] = "x"      -- 1st exp
  t[2] = "y"      -- 2nd exp
  t.x = 1         -- t["x"] = 1
  t[3] = f(x)     -- 3rd exp
  t[30] = 23
  t[4] = 45       -- 4th exp
  a = t
end
```

If the last field in the list has the form `exp` and the expression is a function call or a vararg expression, then all values returned by this expression enter the list consecutively (see §3.4.9).

The field list can have an optional trailing separator, as a convenience for machine-generated code.

3.4.9 · Function Calls

A function call in Lua has the following syntax:

```
functioncall ::= prefixexp args
```

In a function call, first prefixexp and args are evaluated. If the value of prefixexp has type *function*, then this function is called with the given arguments. Otherwise, the prefixexp "call" metamethod is called, having as first parameter the value of prefixexp, followed by the original call arguments (see §2.4).

The form

```
functioncall ::= prefixexp ‘:’ Name args
```

can be used to call "methods". A call `v:name(args)` is syntactic sugar for `v.name(v,args)`, except that `v` is evaluated only once.

Arguments have the following syntax:

```
args ::= ‘(’ [explist] ‘)’
args ::= tableconstructor
args ::= String
```

All argument expressions are evaluated before the call. A call of the form `f{fields}` is syntactic sugar for `f({fields})`; that is, the argument list is a single new table. A call of the form `f'string'` (or `f"string"` or `f[[string]]`) is syntactic sugar for `f('string')`; that is, the argument list is a single literal string.

A call of the form `return functioncall` is called a *tail call*. Lua implements *proper tail calls* (or *proper tail recursion*): in a tail call, the called function reuses the stack entry of the calling function. Therefore, there is no limit on the number of nested tail calls that a program can execute. However, a tail call erases any debug information about the calling function. Note that a tail call only happens with a particular syntax, where the `return` has one single function call as argument; this syntax makes the calling function return exactly the returns of the called function. So, none of the following examples are tail calls:

```
return (f(x))     -- results adjusted to 1
return 2*f(x)
return x,f(x)     -- additional results
f(x); return      -- results discarded
return x or f(x)  -- results adjusted to 1
```

3.4.10 · Function Definitions

The syntax for function definition is

```
functiondef ::= function funcbody
funcbody ::= ‘(’ [parlist] ‘)’ block end
```

The following syntactic sugar simplifies function definitions:

```
stat ::= function funcname funcbody
stat ::= local function Name funcbody
funcname ::= Name {‘.’ Name} [‘:’ Name]
```

The statement

```
function f () body end
```

translates to

```
f = function () body end
```

The statement

```
function t.a.b.c.f () body end
```

translates to

```
t.a.b.c.f = function () body end
```

The statement

```
local function f () body end
```

translates to

```
local f; f = function () body end
```

not to

```
local f = function () body end
```

(This only makes a difference when the body of the function contains references to `f`.)

A function definition is an executable expression, whose value has type *function*. When Lua precompiles a chunk, all its function bodies are precompiled too. Then, whenever Lua executes the function definition, the function is *instantiated* (or *closed*). This function instance (or *closure*) is the final value of the expression.

Parameters act as local variables that are initialized with the argument values:

```
parlist ::= namelist [‘,’ ‘...’] | ‘...’
```

When a function is called, the list of arguments is adjusted to the length of the list of parameters, unless the function is a *vararg function*, which is indicated by three dots (’...’) at the end of its parameter list. A vararg function does not adjust its argument list; instead, it collects all extra arguments and supplies them to the function through a *vararg expression*, which is also written as three dots. The value of this expression is a list of all actual extra arguments, similar to a function with

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

As an example, consider the following definitions:

```
function f(a, b) end
function g(a, b, ...) end
function r() return 1,2,3 end
```

Then we have the following mapping from arguments to parameters and to the vararg expression:

```
CALL            PARAMETERS
------------------------------
f(3)            a=3, b=nil
f(3, 4)         a=3, b=4
f(3, 4, 5)      a=3, b=4
f(r(), 10)      a=1, b=10
f(r())          a=1, b=2
g(3)            a=3, b=nil, ...  --> (nothing)
g(3, 4)         a=3, b=4,   ...  --> (nothing)
g(3, 4, 5, 8)   a=3, b=4,   ...  --> 5  8
g(5, r())       a=5, b=1,   ...  --> 2  3
```

Results are returned using the `return` statement (see §3.3.4). If control reaches the end of a function without encountering a `return` statement, then the function returns with no results.

There is a system-dependent limit on the number of values that a function may return. This limit is guaranteed to be larger than 1000.

The colon syntax is used for defining *methods*, that is, functions that have an implicit extra parameter `self`. Thus, the statement

```
function t.a.b.c:f (params) body end
```

is syntactic sugar for

```
t.a.b.c.f = function (self, params) body end
```

3.5 · Visibility Rules

Lua is a lexically scoped language. The scope of a local variable begins at the first statement after its declaration and lasts until the last non-void statement of the innermost block that includes the declaration. Consider the following example:

```
x = 10                -- global variable
do                    -- new block
  local x = x         -- new 'x', with value 10
  print(x)            --> 10
  x = x+1
  do                  -- another block
    local x = x+1     -- another 'x'
    print(x)          --> 12
  end
  print(x)            --> 11
end
print(x)              --> 10  (the global one)
```

Notice that, in a declaration like `local x = x`, the new `x` being declared is not in scope yet, and so the second `x` refers to the outside variable.

Because of the lexical scoping rules, 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.

Notice that each execution of a `local` statement defines new local variables. Consider the following example:

```
a = {}
local x = 20
for i=1,10 do
  local y = 0
  a[i] = function () y=y+1; return x+y end
end
```

The loop creates ten closures (that is, ten instances of the anonymous function). Each of these closures uses a different `y` variable, while all of them share the same `x`.

4 · The Application Program Interface

This section describes the C API for Lua, that is, the set of C functions available to the host program to communicate with Lua. All API functions and related types and constants are declared in the header file `lua.h`.

Even when we use the term "function", any facility in the API may be provided as a macro instead. Except where stated otherwise, all such macros use each of their arguments exactly once (except for the first argument, which is always a Lua state), and so do not generate any hidden side-effects.

As in most C libraries, the Lua API functions do not check their arguments for validity or consistency. However, you can change this behavior by compiling Lua with the macro `LUA_USE_APICHECK` defined.

4.1 · The Stack

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

Whenever Lua calls C, the called function gets a new stack, which is independent of previous stacks and of stacks of C functions that are still active. This stack initially contains any arguments to the C function and it is where the C function pushes its results to be returned to the caller (see `lua_CFunction`).

For convenience, most query operations in the API do not follow a strict stack discipline. Instead, they 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 relative to the top of the stack. More specifically, if the stack has n elements, then index 1 represents the first element (that is, the element that was pushed onto the stack first) and index n represents the last element; index -1 also represents the last element (that is, the element at the top) and index $-n$ represents the first element.

4.2 · Stack Size

When you interact with the Lua API, you are responsible for ensuring consistency. In particular, *you are responsible for controlling stack overflow*. You can use the function `lua_checkstack` to ensure that the stack has extra slots when pushing new elements.

Whenever Lua calls C, it ensures that the stack has at least `LUA_MINSTACK` extra slots. `LUA_MINSTACK` is defined as 20, so that usually you do not have to worry about stack space unless your code has loops pushing elements onto the stack.

When you call a Lua function without a fixed number of results (see `lua_call`), Lua ensures that the stack has enough size for all results, but it does not ensure any extra space. So, before pushing anything in the stack after such a call you should use `lua_checkstack`.

4.3 · Valid and Acceptable Indices

Any function in the API that receives stack indices works only with *valid indices* or *acceptable indices*.

A *valid index* is an index that refers to a valid position within the stack, that is, it lies between 1 and the stack top ($1 \le |index| \le top$).

Usually, functions that need a specific stack position (e.g., `lua_remove`) require valid indices.

Functions that do not need a specific stack position, but only a value in the stack (e.g., query functions), can be called with acceptable indices. An *acceptable index* refers to a position within the space allocated for the stack, that is, indices up to the stack size. More formally, we define an acceptable index as follows:

```
(index > 0 && abs(index) <= top) ||
(index > 0 && index <= stack size)
```

(Note that 0 is never an acceptable index.) When a function is called, its stack size is `top + LUA_MINSTACK`. You can change its stack size through function `lua_checkstack`.

Acceptable indices serve to avoid extra tests against the stack top when querying the stack. For instance, a C function can query its third argument without the need to first check whether there is a third argument, that is, without the need to check whether 3 is a valid index.

For functions that can be called with acceptable indices, any non-valid index is treated as if it contains a value of a virtual type `LUA_TNONE`.

Unless otherwise noted, any function that accepts valid indices also accepts *pseudo-indices*, which represent some Lua values that are accessible to C code but which are not in the stack. Pseudo-indices are used to access the registry and the upvalues of a C function (see §4.4).

4.4 · C Closures

When a C function is created, it is possible to associate some values with it, thus creating a *C closure* (see `lua_pushcclosure`); these values are called *upvalues* and are accessible to the function whenever it is called.

Whenever a C function is called, its upvalues are located at specific pseudo-indices. These pseudo-indices are produced by the macro `lua_upvalueindex`. The first value associated with a function is at position `lua_upvalueindex(1)`, and so on. Any access to `lua_upvalueindex($n$)`, where n is greater than the number of upvalues of the current function (but not greater than 256), produces an acceptable (but invalid) index.

4.5 · Registry

Lua provides a *registry*, a predefined table that can be used by any C code to store whatever Lua values it needs to store. The registry table is always located at pseudo-index `LUA_REGISTRYINDEX`. Any C library can store data into this table, but it should take care to choose keys that are different from those used by other libraries, to avoid collisions. Typically, you should use as key a string containing your library name, or a light userdata with the address of a C object in your code, or any Lua object created by your code. As with global names, string keys starting with an underscore followed by uppercase letters are reserved for Lua.

The integer keys in the registry are used by the reference mechanism, implemented by the auxiliary library, and by some predefined values. Therefore, integer keys should not be used for other purposes.

When you create a new Lua state, its registry comes with some predefined values. These predefined values are indexed with integer keys defined as constants in `lua.h`. The following constants are defined:

- `LUA_RIDX_MAINTHREAD`: At this index the registry has the main thread of the state. (The main thread is the one created together with the state.)
- `LUA_RIDX_GLOBALS`: At this index the registry has the global environment.

4.6 · Error Handling in C

Internally, Lua uses the C `longjmp` facility to handle errors. (You can also choose to use exceptions if you use C++; see file `luaconf.h`.) When Lua faces any error (such as a memory allocation error, type errors, syntax errors, and runtime errors) it *raises* an error; that is, it does a long jump. A *protected environment* uses `setjmp` to set a recovery point; any error jumps to the most recent active recovery point.

If an error happens outside any protected environment, Lua calls a *panic function* (see `lua_atpanic`) and then calls `abort`, thus exiting the host application. Your panic function can avoid this exit by never returning (e.g., doing a long jump to your own recovery point outside Lua).

The panic function runs as if it were a message handler (see §2.3); in particular, the error message is at the top of the stack. However, there is no guarantees about stack space. To push anything on the stack, the panic function should first check the available space (see §4.2).

Most functions in the API can throw an error, for instance due to a memory allocation error. The documentation for each function indicates whether it can throw errors.

Inside a C function you can throw an error by calling `lua_error`.

4.7 · Handling Yields in C

Internally, Lua uses the C `longjmp` facility to yield a coroutine. Therefore, if a function `foo` calls an API function and this API function yields (directly or indirectly by calling another function that yields), Lua cannot return to `foo` any more, because the `longjmp` removes its frame from the C stack.

To avoid this kind of problem, Lua raises an error whenever it tries to yield across an API call, except for three functions: `lua_yieldk`, `lua_callk`, and `lua_pcallk`. All those functions receive a *continuation function* (as a parameter called `k`) to continue execution after a yield.

We need to set some terminology to explain continuations. We have a C function called from Lua which we will call the *original function*. This original function then calls one of those three functions in the C API, which we will call the *callee function*, that then yields the current thread. (This can happen when the callee function is `lua_yieldk`, or when the callee function is either `lua_callk` or `lu_pcallk` and the function called by them yields.)

Suppose the running thread yields while executing the callee function. After the thread resumes, it eventually will finish running the callee function. However, the callee function cannot return to the original function, because its frame in the C stack was destroyed by the yield. Instead, Lua calls a *continuation function*, which was given as an argument to the callee function. As the name implies, the continuation function should continue the task of the original function.

Lua treats the continuation function as if it were the original function. The continuation function receives the same Lua stack from the original function, in the same state it would be if the callee function had returned. (For instance, after a `lua_callk` the function and its arguments are removed from the stack and replaced by the results from the call.) It also has the same upvalues. Whatever it returns is handled by Lua as if it were the return of the original function.

The only difference in the Lua state between the original function and its continuation is the result of a call to `lua_getctx`.

4.8 · Functions and Types

Here we list all functions and types from the C API in alphabetical order. Each function has an indicator like this: $[-o, +p, x]$

The first field, o, is how many elements the function pops from the stack. The second field, p, is how many elements the function pushes onto the stack. (Any function always pushes its results after popping its arguments.) A field in the form $x|y$ means the function can push (or pop) x or y elements, depending on the situation; an interrogation mark ? means that we cannot know how many elements the function pops/pushes by looking only at its arguments (e.g., they may depend on what is on the stack). The third field, x, tells whether the function may throw errors: $-$ means the function never throws any error; e means the function may throw errors; v means the function may throw an error on purpose.

`lua_absindex` $[-0, +0, -]$

```
int lua_absindex (lua_State *L, int idx);
```

Converts the acceptable index `idx` into an absolute index (that is, one that does not depend on the stack top).

`lua_Alloc`

```
typedef void * (*lua_Alloc) (void *ud,
                             void *ptr,
                             size_t osize,
                             size_t nsize);
```

The type of the memory-allocation function used by Lua states. The allocator function must provide a functionality similar to `realloc`, but not exactly the same. Its arguments are `ud`, an opaque pointer passed to `lua_newstate`; `ptr`, a pointer to the block being allocated/reallocated/freed; `osize`, the original size of the block or some code about what is being allocated; `nsize`, the new size of the block.

When `ptr` is not `NULL`, `osize` is the size of the block pointed by `ptr`, that is, the size given when it was allocated or reallocated.

When `ptr` is `NULL`, `osize` encodes the kind of object that Lua is allocating. `osize` is any of `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, or `LUA_TTHREAD` when (and only when) Lua is creating a new object of that type. When `osize` is some other value, Lua is allocating memory for something else.

Lua assumes the following behavior from the allocator function:

When `nsize` is zero, the allocator should behave like `free` and return `NULL`.

When `nsize` is not zero, the allocator should behave like `realloc`. The allocator returns `NULL` if and only if it cannot fulfill the request. Lua assumes that the allocator never fails when `osize >= nsize`.

Here is a simple implementation for the allocator function. It is used in the auxiliary library by `luaL_newstate`.

```
static void *l_alloc
    (void *ud, void *ptr, size_t osize,
     size_t nsize) {
  (void)ud;  (void)osize;  /* not used */
  if (nsize == 0) {
    free(ptr);
    return NULL;
  }
  else
    return realloc(ptr, nsize);
}
```

Note that Standard C ensures that `free(NULL)` has no effect and that `realloc(NULL, size)` is equivalent to `malloc(size)`. This code assumes that `realloc` does not fail when shrinking a block. (Although Standard C does not ensure this behavior, it seems to be a safe assumption.)

`lua_arith` $[-(2|1), +1, e]$

```
void lua_arith (lua_State *L, int op);
```

Performs an arithmetic operation over the two values (or one, in the case of negation) at the top of the stack, with the value at the top being the second operand, pops these values, and pushes the result of the operation. The function follows the semantics of the corresponding Lua operator (that is, it may call metamethods).

The value of `op` must be one of the following constants:

- `LUA_OPADD`: addition (`+`)
- `LUA_OPSUB`: subtraction (`-`)
- `LUA_OPMUL`: multiplication (`*`)
- `LUA_OPDIV`: division (`/`)
- `LUA_OPMOD`: modulo (`%`)
- `LUA_OPPOW`: exponentiation (`^`)
- `LUA_OPUNM`: arithmetic negation (unary `-`)

`lua_atpanic` $[-0, +0, -]$

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

Sets a new panic function and returns the old one (see §4.6).

`lua_call` $[-(nargs + 1), +nresults, e]$

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

Calls a function.

To call a function you must use the following protocol: first, the function to be called is pushed onto the stack; then, the arguments to the function are pushed in direct order; that is, the first argument is pushed first. Finally you call `lua_call`; `nargs` is the number of arguments that you pushed onto the stack. All arguments and the function value are popped from the stack when the function is called. The function results are pushed onto the stack when the function returns. The number of results is adjusted to `nresults`, unless `nresults` is `LUA_MULTRET`. In this case, all results from the function are pushed. Lua takes care that the returned values fit into the stack space. The function results are pushed onto the stack in direct order (the first result is pushed first), so that after the call the last result is on the top of the stack.

Any error inside the called function is propagated upwards (with a `longjmp`).

The following example shows how the host program can do the equivalent to this Lua code:

```
a = f("how", t.x, 14)
```

Here it is in C:

```
lua_getglobal(L,"f");    // f is to be called
lua_pushstring(L,"how"); // 1st argument
lua_getglobal(L,"t");    // t is to be indexed
lua_getfield(L,-1,"x");  // push result of t.x
                         // (2nd arg)
lua_remove(L,-2);        // remove 't' from the
                         // stack
lua_pushinteger(L,14);   // 3rd argument
lua_call(L,3,1);         // call 'f' with 3
                         // args and 1 result
lua_setglobal(L,"a");    // set global 'a'
```

Note that the code above is "balanced": at its end, the stack is back to its original configuration. This is considered good programming practice.

`lua_callk` $[-(nargs + 1), +nresults, e]$

```
void lua_callk (lua_State *L, int nargs,
                int nresults, int ctx,
                lua_CFunction k);
```

This function behaves exactly like `lua_call`, but allows the called function to yield (see §4.7).

`lua_CFunction`

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

Type for C functions.

In order to communicate properly with Lua, a C function must use the following protocol, which defines the way parameters and results are passed: a C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first). So, when the function starts, `lua_gettop(L)` returns the number of arguments received by the function. The first argument (if any) is at index 1 and its last argument is at index `lua_gettop(L)`. To return values to Lua, a C function just pushes them onto the stack, in direct order (the first result is pushed first), and returns the number of results. Any other value in the stack below the results will be properly discarded by Lua. Like a Lua function, a C function called by Lua can also return many results.

As an example, the following function receives a variable number of numerical arguments and returns their average and sum:

```
static int foo (lua_State *L) {
  int n = lua_gettop(L);  // number of args
  lua_Number sum = 0;
  int i;
  for (i = 1; i <= n; ++i) {
    if (!lua_isnumber(L,i)) {
      lua_pushstring(L,"incorrect argument");
      lua_error(L);
    }
    sum += lua_tonumber(L,i);
  }
  lua_pushnumber(L,sum/n);  // first result
  lua_pushnumber(L,sum);    // second result
  return 2;             // number of results
}
```

`lua_checkstack` $[-0, +0, -]$

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

Ensures that there are at least `extra` free stack slots in the stack. It returns false if it cannot fulfill the request, because it would cause the stack to be larger than a fixed maximum size (typically at least a few thousand elements) or because it cannot allocate memory for the new stack size. This function never shrinks the stack; if the stack is already larger than the new size, it is left unchanged.

`lua_close` $[-0, +0, -]$

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

Destroys all objects in the given Lua state (calling the corresponding garbage-collection metamethods, if any) and frees all dynamic memory used by this state. On several platforms, you may not need to call this function, because all resources are naturally released when the host program ends. On the other hand, long-running programs that create multiple states, such as daemons or web servers, might need to close states as soon as they are not needed.

`lua_compare` $[-0, +0, e]$

```
int lua_compare (lua_State *L, int index1,
                 int index2, int op);
```

Compares two Lua values. Returns 1 if the value at acceptable index `index1` satisfies `op` when compared with the value at acceptable index `index2`, following the semantics of the corresponding Lua operator (that is, it may call metamethods). Otherwise returns 0. Also returns 0 if any of the indices is non valid.

The value of `op` must be one of the following constants:

- `LUA_OPEQ`: compares for equality (`==`)
- `LUA_OPLT`: compares for less than (`<`)
- `LUA_OPLE`: compares for less or equal (`<=`)

`lua_concat` $[-n, +1, e]$

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

Concatenates the `n` values at the top of the stack, pops them, and leaves the result at the top. If `n` is 1, the result is the single value on the stack (that is, the function does nothing); if `n` is 0, the result is the empty string. Concatenation is performed following the usual semantics of Lua (see §3.4.5).

`lua_copy` $[-0, +0, -]$

```
void lua_copy (lua_State *L, int fromidx,
               int toidx);
```

Moves the element at the valid index `fromidx` into the valid index `toidx` without shifting any element (therefore replacing the value at that position).

`lua_createtable` $[-0, +1, e]$

```
void lua_createtable (lua_State *L, int narr,
                      int nrec);
```

Creates a new empty table and pushes it onto the stack. Parameter `narr` is a hint for how many elements the table will have as a sequence; parameter `nrec` is a hint for how many other elements the table will have. Lua may use these hints to preallocate memory for the new table. This pre-allocation is useful for performance when you know in advance how many elements the table will have. Otherwise you can use the function `lua_newtable`.

`lua_dump` $[-0, +0, e]$

```
int lua_dump (lua_State *L, lua_Writer writer,
              void *data);
```

Dumps a function as a binary chunk. Receives a Lua function on the top of the stack and produces a binary chunk that, if loaded again, results in a function equivalent to the one dumped. As it produces parts of the chunk, `lua_dump` calls function `writer` (see `lua_Writer`) with the given `data` to write them.

The value returned is the error code returned by the last call to the writer; 0 means no errors.

This function does not pop the Lua function from the stack.

`lua_error` $[-1, +0, v]$

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

Generates a Lua error. The error message (which can actually be a Lua value of any type) must be on the stack top. This function does a long jump, and therefore never returns (see `luaL_error`).

`lua_gc` $[-0, +0, e]$

```
int lua_gc (lua_State *L, int what, int data);
```

Controls the garbage collector.

This function performs several tasks, according to the value of the parameter `what`:

- `LUA_GCSTOP`: stops the garbage collector.
- `LUA_GCRESTART`: restarts the garbage collector.
- `LUA_GCCOLLECT`: performs a full garbage-collection cycle.
- `LUA_GCCOUNT`: returns the current amount of memory (in Kbytes) in use by Lua.
- `LUA_GCCOUNTB`: returns the remainder of dividing the current amount of bytes of memory in use by Lua by 1024.
- `LUA_GCSTEP`: performs an incremental step of garbage collection. The step "size" is controlled by `data` (larger values mean more steps) in a non-specified way. If you want to control the step size you must experimentally tune the value of `data`. The function returns 1 if the step finished a garbage-collection cycle.
- `LUA_GCSETPAUSE`: sets `data` as the new value for the *pause* of the collector (see §2.5). The function returns the previous value of the pause.

- LUA_GCSETSTEPMUL: sets data as the new value for the *step multiplier* of the collector (see §2.5). The function returns the previous value of the step multiplier.
- LUA_GCISRUNNING: returns a boolean that tells whether the collector is running (i.e., not stopped).
- LUA_GCGEN: changes the collector to generational mode (see §2.5).
- LUA_GCINC: changes the collector to incremental mode. This is the default mode.

For more details about these options, see collectgarbage.

---

lua_getallocf $[-0, +0, -]$

```
lua_Alloc lua_getallocf (lua_State *L,
                         void **ud);
```

Returns the memory-allocation function of a given state. If ud is not NULL, Lua stores in *ud the opaque pointer passed to lua_newstate.

---

lua_getctx $[-0, +0, -]$

```
int lua_getctx (lua_State *L, int *ctx);
```

This function is called by a continuation function (see §4.7) to retrieve the status of the thread and a context information.

When called in the original function, lua_getctx always returns LUA_OK and does not change the value of its argument ctx. When called inside a continuation function, lua_getctx returns LUA_YIELD and sets the value of ctx to be the context information (the value passed as the ctx argument to the callee together with the continuation function).

When the callee is lua_pcallk, Lua may also call its continuation function to handle errors during the call. That is, upon an error in the function called by lua_pcallk, Lua may not return to the original function but instead may call the continuation function. In that case, a call to lua_getctx will return the error code (the value that would be returned by lua_pcallk); the value of ctx will be set to the context information, as in the case of a yield.

---

lua_getfield $[-0, +1, e]$

```
void lua_getfield (lua_State *L, int index,
                   const char *k);
```

Pushes onto the stack the value t[k], where t is the value at the given valid index. As in Lua, this function may trigger a metamethod for the "index" event (see §2.4).

---

lua_getglobal $[-0, +1, e]$

```
void lua_getglobal (lua_State *L,
                    const char *name);
```

Pushes onto the stack the value of the global name.

---

lua_getmetatable $[-0, +(0|1), -]$

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

Pushes onto the stack the metatable of the value at the given acceptable index. If the value does not have a metatable, the function returns 0 and pushes nothing on the stack.

---

lua_gettable $[-1, +1, e]$

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

Pushes onto the stack the value t[k], where t is the value at the given valid index and k is the value at the top of the stack.

This function pops the key from the stack (putting the resulting value in its place). As in Lua, this function may trigger a metamethod for the "index" event (see §2.4).

---

lua_gettop $[-0, +0, -]$

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

Returns the index of the top element in the stack. Because indices start at 1, this result is equal to the number of elements in the stack (and so 0 means an empty stack).

---

lua_getuservalue $[-0, +1, -]$

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

Pushes onto the stack the Lua value associated with the userdata at the given index. This Lua value must be a table or nil.

---

lua_insert $[-1, +1, -]$

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

Moves the top element into the given valid index, shifting up the elements above this index to open space. Cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.

---

lua_Integer

```
typedef ptrdiff_t lua_Integer;
```

The type used by the Lua API to represent signed integral values.

By default it is a ptrdiff_t, which is usually the largest signed integral type the machine handles "comfortably".

---

lua_isboolean $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a boolean, and 0 otherwise.

`lua_iscfunction` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a C function, and 0 otherwise.

`lua_isfunction` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a function (either C or Lua), and 0 otherwise.

`lua_islightuserdata` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a light userdata, and 0 otherwise.

`lua_isnil` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is `nil`, and 0 otherwise.

`lua_isnone` $[-0, +0, -]$

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

Returns 1 if the given acceptable index is not valid (that is, it refers to an element outside the current stack), and 0 otherwise.

`lua_isnoneornil` $[-0, +0, -]$

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

Returns 1 if the given acceptable index is not valid (that is, it refers to an element outside the current stack) or if the value at this index is `nil`, and 0 otherwise.

`lua_isnumber` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a number or a string convertible to a number, and 0 otherwise.

`lua_isstring` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a string or a number (which is always convertible to a string), and 0 otherwise.

`lua_istable` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a table, and 0 otherwise.

`lua_isthread` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a thread, and 0 otherwise.

`lua_isuserdata` $[-0, +0, -]$

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

Returns 1 if the value at the given acceptable index is a userdata (either full or light), and 0 otherwise.

`lua_len` $[-0, +1, e]$

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

Returns the "length" of the value at the given acceptable index; it is equivalent to the '`#`' operator in Lua (see §3.4.6). The result is pushed on the stack.

`lua_load` $[-0, +1, -]$

```
int lua_load (lua_State *L,
              lua_Reader reader,
              void *data,
              const char *source,
              const char *mode);
```

Loads a Lua chunk (without running it). If there are no errors, `lua_load` pushes the compiled chunk as a Lua function on top of the stack. Otherwise, it pushes an error message.

The return values of `lua_load` are:

- `LUA_OK`: no errors;
- `LUA_ERRSYNTAX`: syntax error during precompilation;
- `LUA_ERRMEM`: memory allocation error;
- `LUA_ERRGCMM`: error while running a `__gc` metamethod. (This error has no relation with the chunk being loaded. It is generated by the garbage collector.)

The `lua_load` function uses a user-supplied `reader` function to read the chunk (see `lua_Reader`). The `data` argument is an opaque value passed to the reader function.

The `source` argument gives a name to the chunk, which is used for error messages and in debug information (see §4.9).

`lua_load` automatically detects whether the chunk is text or binary and loads it accordingly (see program `luac`). The string `mode` works as in function `load`, with the addition that a `NULL` value is equivalent to the string `"bt"`.

If the resulting function has one upvalue, this upvalue is set to the value of the global environment stored at index `LUA_RIDX_GLOBALS` in the registry (see §4.5). When loading main chunks, this upvalue will be the `_ENV` variable (see §2.2).

`lua_newstate` $[-0, +0, -]$

```
lua_State *lua_newstate (lua_Alloc f, void *ud);
```

Creates a new thread running in a new, independent state. Returns `NULL` if cannot create the thread or the state (due to lack of memory). The argument `f` is the allocator function; Lua does all memory allocation for this state through this function. The second argument, `ud`, is an opaque pointer that Lua passes to the allocator in every call.

`lua_newtable` $[-0, +1, e]$

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

Creates a new empty table and pushes it onto the stack. It is equivalent to `lua_createtable(L, 0, 0)`.

`lua_newthread` $[-0, +1, e]$

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

Creates a new thread, pushes it on the stack, and returns a pointer to a `lua_State` that represents this new thread. The new thread returned by this function shares with the original thread its global environment, but has an independent execution stack.

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

`lua_newuserdata` $[-0, +1, e]$

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

This function allocates a new block of memory with the given size, pushes onto the stack a new full userdata with the block address, and returns this address. The host program can freely use this memory.

`lua_next` $[-1, +(2|0), e]$

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

Pops a key from the stack, and pushes a key—value pair from the table at the given index (the "next" pair after the given key). If there are no more elements in the table, then `lua_next` returns 0 (and pushes nothing).

A typical traversal looks like this:

```
/* table is in the stack at index 't' */
lua_pushnil(L);  /* first key */
while (lua_next(L, t) != 0) {
  /* uses 'key' (at index -2) and 'value'
     (at index -1) */
  printf("%s - %s\n",
         lua_typename(L, lua_type(L, -2)),
         lua_typename(L, lua_type(L, -1)));
  /* removes 'value'; keeps 'key' for next
     iteration */
  lua_pop(L, 1);
}
```

While traversing a table, do not call `lua_tolstring` directly on a key, unless you know that the key is actually a string. Recall that `lua_tolstring` may change the value at the given index; this confuses the next call to `lua_next`.

See function `next` for the caveats of modifying the table during its traversal.

`lua_Number`

```
typedef double lua_Number;
```

The type of numbers in Lua. By default, it is double, but that can be changed in `luaconf.h`. Through this configuration file you can change Lua to operate with another type for numbers (e.g., float or long).

`lua_pcall` $[-(nargs + 1), +(nresults|1), -]$

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

Calls a function in protected mode.

Both `nargs` and `nresults` have the same meaning as in `lua_call`. If there are no errors during the call, `lua_pcall` behaves exactly like `lua_call`. However, if there is any error, `lua_pcall` catches it, pushes a single value on the stack (the error message), and returns an error code. Like `lua_call`, `lua_pcall` always removes the function and its arguments from the stack.

If `msgh` is 0, then the error message returned on the stack is exactly the original error message. Otherwise, `msgh` is the stack index of a *message handler*. (In the current implementation, this index cannot be a pseudo-index.) In case of runtime errors, this function will be called with the error message and its return value will be the message returned on the stack by `lua_pcall`.

Typically, the message handler is used to add more debug information to the error message, such as a stack traceback. Such information cannot be gathered after the return of `lua_pcall`, since by then the stack has unwound.

The `lua_pcall` function returns one of the following codes (defined in `lua.h`):

- `LUA_OK` (0): success.
- `LUA_ERRRUN`: a runtime error.
- `LUA_ERRMEM`: memory allocation error. For such errors, Lua does not call the message handler.
- `LUA_ERRERR`: error while running the message handler.

- LUA_ERRGCMM: error while running a __gc metamethod. (This error typically has no relation with the function being called. It is generated by the garbage collector.)

---

lua_pcallk $[-(nargs+1), +(nresults|1), -]$

```
int lua_pcallk (lua_State *L,
                int nargs,
                int nresults,
                int errfunc,
                int ctx,
                lua_CFunction k);
```

This function behaves exactly like lua_pcall, but allows the called function to yield (see §4.7).

---

lua_pop $[-n, +0, -]$

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

Pops n elements from the stack.

---

lua_pushboolean $[-0, +1, -]$

```
void lua_pushboolean (lua_State *L, int b);
```

Pushes a boolean value with value b onto the stack.

---

lua_pushcclosure $[-n, +1, e]$

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

Pushes a new C closure onto the stack.

When a C function is created, it is possible to associate some values with it, thus creating a C closure (see §4.4); these values are then accessible to the function whenever it is called. To associate values with a C function, first these values should be pushed onto the stack (when there are multiple values, the first value is pushed first). Then lua_pushcclosure is called to create and push the C function onto the stack, with the argument n telling how many values should be associated with the function. lua_pushcclosure also pops these values from the stack.

The maximum value for n is 255.

When n is zero, this function creates a *light C function*, which is just a pointer to the C function. In that case, it never throws a memory error.

---

lua_pushcfunction $[-0, +1, -]$

```
void lua_pushcfunction (lua_State *L,
                        lua_CFunction f);
```

Pushes a C function onto the stack. This function receives a pointer to a C function and pushes onto the stack a Lua value of type *function* that, when called, invokes the corresponding C function.

Any function to be registered in Lua must follow the correct protocol to receive its parameters and return its results (see lua_CFunction).

lua_pushcfunction is defined as a macro:

```
#define lua_pushcfunction(L,f) \
        lua_pushcclosure(L,f,0)
```

Note that f is used twice.

---

lua_pushfstring $[-0, +1, e]$

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

Pushes onto the stack a formatted string and returns a pointer to this string. It is similar to the C function sprintf, but has some important differences:

- You do not have to allocate space for the result: the result is a Lua string and Lua takes care of memory allocation (and deallocation, through garbage collection).
- The conversion specifiers are quite restricted. There are no flags, widths, or precisions. The conversion specifiers can only be %% (inserts a '%' in the string), %s (inserts a zero-terminated string, with no size restrictions), %f (inserts a lua_Number), %p (inserts a pointer as a hexadecimal numeral), %d (inserts an int), and %c (inserts an int as a byte).

---

lua_pushinteger $[-0, +1, -]$

```
void lua_pushinteger (lua_State *L,
                      lua_Integer n);
```

Pushes a number with value n onto the stack.

---

lua_pushlightuserdata $[-0, +1, -]$

```
void lua_pushlightuserdata (lua_State *L,
                            void *p);
```

Pushes a light userdata onto the stack.

Userdata represent C values in Lua. A *light userdata* represents a pointer, a void*. It is a value (like a number): you do not create it, it has no individual metatable, and it is not collected (as it was never created). A light userdata is equal to "any" light userdata with the same C address.

---

lua_pushliteral $[-0, +1, e]$

```
const char *lua_pushliteral (lua_State *L,
                             const char *s);
```

This macro is equivalent to lua_pushlstring, but can be used only when s is a literal string. It automatically provides the string length.

---

lua_pushlstring $[-0, +1, e]$

```
const char *lua_pushlstring
     (lua_State *L, const char *s, size_t len);
```

Pushes the string pointed to by `s` with size `len` onto the stack. Lua makes (or reuses) an internal copy of the given string, so the memory at `s` can be freed or reused immediately after the function returns. The string can contain any binary data, including embedded zeros.

Returns a pointer to the internal copy of the string.

`lua_pushnil` $[-0, +1, -]$

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

Pushes a nil value onto the stack.

`lua_pushnumber` $[-0, +1, -]$

```
void lua_pushnumber (lua_State *L,lua_Number n);
```

Pushes a number with value `n` onto the stack.

`lua_pushstring` $[-0, +1, e]$

```
const char *lua_pushstring (lua_State *L,
                            const char *s);
```

Pushes the zero-terminated string pointed to by `s` onto the stack. Lua makes (or reuses) an internal copy of the given string, so the memory at `s` can be freed or reused immediately after the function returns.

Returns a pointer to the internal copy of the string.

If `s` is `NULL`, pushes `nil` and returns `NULL`.

`lua_pushthread` $[-0, +1, -]$

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

Pushes the thread represented by `L` onto the stack. Returns 1 if this thread is the main thread of its state.

`lua_pushvalue` $[-0, +1, -]$

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

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

`lua_pushvfstring` $[-0, +1, e]$

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

Equivalent to `lua_pushfstring`, except that it receives a `va_list` instead of a variable number of arguments.

`lua_rawequal` $[-0, +0, -]$

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

Returns 1 if the two values in acceptable indices `index1` and `index2` are primitively equal (that is, without calling metamethods). Otherwise returns 0. Also returns 0 if any of the indices are non valid.

`lua_rawget` $[-1, +1, -]$

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

Similar to `lua_gettable`, but does a raw access (i.e., without metamethods).

`lua_rawgeti` $[-0, +1, -]$

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

Pushes onto the stack the value `t[n]`, where `t` is the table at the given valid index. The access is raw; that is, it does not invoke metamethods.

`lua_rawgetp` $[-0, +1, -]$

```
void lua_rawgetp (lua_State *L, int index,
                  const void *p);
```

Pushes onto the stack the value `t[k]`, where `t` is the table at the given valid index and `k` is the pointer `p` represented as a light userdata. The access is raw; that is, it does not invoke metamethods.

`lua_rawlen` $[-0, +0, -]$

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

Returns the raw "length" of the value at the given acceptable index: for strings, this is the string length; for tables, this is the result of the length operator ('`#`') with no metamethods; for userdata, this is the size of the block of memory allocated for the userdata; for other values, it is 0.

`lua_rawset` $[-2, +0, e]$

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

Similar to `lua_settable`, but does a raw assignment (i.e., without metamethods).

`lua_rawseti` $[-1, +0, e]$

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

Does the equivalent of `t[n] = v`, where `t` is the table at the given valid index and `v` is the value at the top of the stack.

This function pops the value from the stack. The assignment is raw; that is, it does not invoke metamethods.

`lua_rawsetp` $[-1, +0, e]$

```
void lua_rawsetp (lua_State *L, int index,
                  const void *p);
```

Does the equivalent of `t[k] = v`, where `t` is the table at the given valid index, `k` is the pointer `p` represented as a light userdata, and `v` is the value at the top of the stack.

This function pops the value from the stack. The assignment is raw; that is, it does not invoke metamethods.

`lua_Reader`

```
typedef const char * (*lua_Reader)
     (lua_State *L, void *data, size_t *size);
```

The reader function used by `lua_load`. Every time it needs another piece of the chunk, `lua_load` calls the reader, passing along its `data` parameter. The reader must return a pointer to a block of memory with a new piece of the chunk and set `size` to the block size. The block must exist until the reader function is called again. To signal the end of the chunk, the reader must return `NULL` or set `size` to zero. The reader function may return pieces of any size greater than zero.

`lua_register` $[-0, +0, e]$

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

Sets the C function `f` as the new value of global `name`. It is defined as a macro:

```
#define lua_register(L,n,f) \
        (lua_pushcfunction(L,f), \
         lua_setglobal(L, n))
```

`lua_remove` $[-1, +0, -]$

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

Removes the element at the given valid index, shifting down the elements above this index to fill the gap. Cannot be called with a pseudo-index, because a pseudo-index is not an actual stack position.

`lua_replace` $[-1, +0, -]$

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

Moves the top element into the given position without shifting any element (therefore replacing the value at the given position), and then pops the top element.

`lua_resume` $[-?, +?, -]$

```
int lua_resume (lua_State *L, lua_State *from,
                int nargs);
```

Starts and resumes a coroutine in a given thread.

To start a coroutine, you push onto the thread stack the main function plus any arguments; then you call `lua_resume`, with `nargs` being the number of arguments. This call returns when the coroutine suspends or finishes its execution. When it returns, the stack contains all values passed to `lua_yield`, or all values returned by the body function. `lua_resume` returns `LUA_YIELD` if the coroutine yields, `LUA_OK` if the coroutine finishes its execution without errors, or an error code in case of errors (see `lua_pcall`).

In case of errors, the stack is not unwound, so you can use the debug API over it. The error message is on the top of the stack.

To resume a coroutine, you put on its stack only the values to be passed as results from `yield`, and then call `lua_resume`.

The parameter `from` represents the coroutine that is resuming `L`. If there is no such coroutine, this parameter can be `NULL`.

`lua_setallocf` $[-0, +0, -]$

```
void lua_setallocf (lua_State *L, lua_Alloc f,
                    void *ud);
```

Changes the allocator function of a given state to `f` with user data `ud`.

`lua_setfield` $[-1, +0, e]$

```
void lua_setfield (lua_State *L, int index,
                   const char *k);
```

Does the equivalent to `t[k] = v`, where `t` is the value at the given valid index and `v` is the value at the top of the stack.

This function pops the value from the stack. As in Lua, this function may trigger a metamethod for the "newindex" event (see §2.4).

`lua_setglobal` $[-1, +0, e]$

```
void lua_setglobal (lua_State *L,
                    const char *name);
```

Pops a value from the stack and sets it as the new value of global `name`.

`lua_setmetatable` $[-1, +0, -]$

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

Pops a table from the stack and sets it as the new metatable for the value at the given acceptable index.

`lua_settable` $[-2, +0, e]$

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

Does the equivalent to `t[k] = v`, where `t` is the value at the given valid index, `v` is the value at the top of the stack, and `k` is the value just below the top.

This function pops both the key and the value from the stack. As in Lua, this function may trigger a metamethod for the "newindex" event (see §2.4).

`lua_settop` $[-?, +?, -]$

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

Accepts any acceptable index, or 0, and sets the stack top to this 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.

`lua_setuservalue` $[-1, +0, -]$

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

Pops a table or `nil` from the stack and sets it as the new value associated to the userdata at the given index.

`lua_State`

```
typedef struct lua_State lua_State;
```

An opaque structure that points to a thread and indirectly (through the thread) to the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables. All information about a state is accessible through this structure.

A pointer to this structure must be passed as the first argument to every function in the library, except to `lua_newstate`, which creates a Lua state from scratch.

`lua_status` $[-0, +0, -]$

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

Returns the status of the thread `L`.

The status can be 0 (`LUA_OK`) for a normal thread, an error code if the thread finished the execution of a `lua_resume` with an error, or `LUA_YIELD` if the thread is suspended.

You can only call functions in threads with status `LUA_OK`. You can resume threads with status `LUA_OK` (to start a new coroutine) or `LUA_YIELD` (to resume a coroutine).

`lua_toboolean` $[-0, +0, -]$

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

Converts the Lua value at the given acceptable index to a C boolean value (0 or 1). Like all tests in Lua, `lua_toboolean` returns true for any Lua value different from `false` and `nil`; otherwise it returns false. It also returns false when called with a non-valid index. (If you want to accept only actual boolean values, use `lua_isboolean` to test the value's type.)

`lua_tocfunction` $[-0, +0, -]$

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

Converts a value at the given acceptable index to a C function. That value must be a C function; otherwise, returns `NULL`.

`lua_tointeger` $[-0, +0, -]$

```
lua_Integer lua_tointeger (lua_State *L,
                           int index);
```

Equivalent to `lua_tointegerx` with `isnum` equal to `NULL`.

`lua_tointegerx` $[-0, +0, -]$

```
lua_Integer lua_tointegerx
        (lua_State *L, int index, int *isnum);
```

Converts the Lua value at the given acceptable index to the signed integral type `lua_Integer`. The Lua value must be a number or a string convertible to a number (see §3.4.2); otherwise, `lua_tointegerx` returns 0.

If the number is not an integer, it is truncated in some non-specified way.

If `isnum` is not `NULL`, its referent is assigned a boolean value that indicates whether the operation succeeded.

`lua_tolstring` $[-0, +0, e]$

```
const char *lua_tolstring
       (lua_State *L, int index, size_t *len);
```

Converts the Lua value at the given acceptable index to a C string. If `len` is not `NULL`, it also sets `*len` with the string length. The Lua value must be a string or a number; otherwise, the function returns `NULL`. If the value is a number, then `lua_tolstring` also *changes the actual value in the stack to a string.* (This change confuses `lua_next` when `lua_tolstring` is applied to keys during a table traversal.)

`lua_tolstring` returns a fully aligned pointer to a string inside the Lua state. This string always has a zero (`'\0'`) after its last character (as in C), but can contain other zeros in its body. Because Lua has garbage collection, there is no guarantee that the pointer returned by `lua_tolstring` will be valid after the corresponding value is removed from the stack.

`lua_tonumber` $[-0, +0, -]$

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

Equivalent to `lua_tonumberx` with `isnum` equal to `NULL`.

`lua_tonumberx` $[-0, +0, -]$

```
lua_Number lua_tonumberx
        (lua_State *L, int index, int *isnum);
```

Converts the Lua value at the given acceptable index to the C type `lua_Number` (see `lua_Number`). The Lua value must be a number or a string convertible to a number (see §3.4.2); otherwise, `lua_tonumberx` returns 0.

If `isnum` is not `NULL`, its referent is assigned a boolean value that indicates whether the operation succeeded.

`lua_topointer` $[-0, +0, -]$

```
const void *lua_topointer (lua_State *L,
                           int index);
```

Converts the value at the given acceptable index to a generic C pointer (`void*`). The value can be a userdata, a table, a thread, or a function; otherwise, `lua_topointer` returns `NULL`. Different objects will give different pointers. There is no way to convert the pointer back to its original value.

Typically this function is used only for debug information.

`lua_tostring` $[-0, +0, e]$

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

Equivalent to `lua_tolstring` with `len` equal to `NULL`.

`lua_tothread` $[-0, +0, -]$

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

Converts the value at the given acceptable index to a Lua thread (represented as `lua_State*`). This value must be a thread; otherwise, the function returns `NULL`.

`lua_tounsigned` $[-0, +0, -]$

```
lua_Unsigned lua_tounsigned (lua_State *L,
                             int index);
```

Equivalent to `lua_tounsignedx` with `isnum` equal to `NULL`.

`lua_tounsignedx` $[-0, +0, -]$

```
lua_Unsigned lua_tounsignedx
        (lua_State *L, int index, int *isnum);
```

Converts the Lua value at the given acceptable index to the unsigned integral type `lua_Unsigned`. The Lua value must be a number or a string convertible to a number (see §3.4.2); otherwise, `lua_tounsignedx` returns 0.

If the number is not an integer, it is truncated in some non-specified way. If the number is outside the range of representable values, it is normalized to the remainder of its division by one more than the maximum representable value.

If `isnum` is not `NULL`, its referent is assigned a boolean value that indicates whether the operation succeeded.

`lua_touserdata` $[-0, +0, -]$

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

If the value at the given acceptable index is a full userdata, returns its block address. If the value is a light userdata, returns its pointer. Otherwise, returns `NULL`.

`lua_type` $[-0, +0, -]$

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

Returns the type of the value in the given acceptable index, or `LUA_TNONE` for a non-valid index. The types returned by `lua_type` are coded by the following constants defined in `lua.h`: `LUA_TNIL`, `LUA_TNUMBER`, `LUA_TBOOLEAN`, `LUA_TSTRING`, `LUA_TTABLE`, `LUA_TFUNCTION`, `LUA_TUSERDATA`, `LUA_TTHREAD`, and `LUA_TLIGHTUSERDATA`.

`lua_typename` $[-0, +0, -]$

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

Returns the name of the type encoded by the value `tp`, which must be one the values returned by `lua_type`.

`lua_Unsigned`

```
typedef unsigned long lua_Unsigned;
```

The type used by the Lua API to represent unsigned integral values. It must have at least 32 bits.

By default it is an `unsigned int` or an `unsigned long`, whichever can hold 32-bit values.

`lua_version` $[-0, +0, v]$

```
const lua_Number *lua_version (lua_State *L);
```

Returns the address of the version number stored in the Lua core. When called with a valid `lua_State`, returns the address of the version used to create that state. When called with `NULL`, returns the address of the version running the call.

`lua_Writer`

```
typedef int (*lua_Writer) (lua_State *L,
                           const void* p,
                           size_t sz,
                           void* ud);
```

The type of the writer function used by `lua_dump`. Every time it produces another piece of chunk, `lua_dump` calls the writer, passing along the buffer to be written (`p`), its size (`sz`), and the `data` parameter supplied to `lua_dump`.

The writer returns an error code: 0 means no errors; any other value means an error and stops `lua_dump` from calling the writer again.

`lua_xmove` $[-?, +?, -]$

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

Exchange values between different threads of the same state.

This function pops `n` values from the stack `from`, and pushes them onto the stack `to`.

`lua_yield` $[-?, +?, -]$

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

This function is equivalent to `lua_yieldk`, but it has no continuation (see §4.7). Therefore, when the thread resumes, it returns to the function that called the function calling `lua_yield`.

`lua_yieldk` $[-?, +?, -]$

```
int lua_yieldk (lua_State *L, int nresults,
                int ctx, lua_CFunction k);
```

Yields a coroutine.

This function should only be called as the return expression of a C function, as follows:

```
return lua_yieldk (L, n, i, k);
```

When a C function calls `lua_yieldk` in that way, the running coroutine suspends its execution, and the call to `lua_resume` that started this coroutine returns. The parameter `nresults` is the number of values from the stack that are passed as results to `lua_resume`.

When the coroutine is resumed again, Lua calls the given continuation function `k` to continue the execution of the C function that yielded (see §4.7). This continuation function receives the same stack from the previous function, with the results removed and replaced by the arguments passed to `lua_resume`. Moreover, the continuation function may access the value `ctx` by calling `lua_getctx`.

4.9 · The Debug Interface

Lua has no built-in debugging facilities. Instead, it offers a special interface by means of functions and *hooks*. This interface allows the construction of different kinds of debuggers, profilers, and other tools that need "inside information" from the interpreter.

`lua_Debug`

```
typedef struct lua_Debug {
  int event;
  const char *name;           // (n)
  const char *namewhat;       // (n)
  const char *what;           // (S)
  const char *source;         // (S)
  int currentline;            // (l)
  int linedefined;            // (S)
  int lastlinedefined;        // (S)
  unsigned char nups;         // (u) upvalues
  unsigned char nparams;      // (u) parameters
  char isvararg;              // (u)
  char istailcall;            // (t)
  char short_src[LUA_IDSIZE]; // (S)
  /* private part */
  other fields
} lua_Debug;
```

A structure used to carry different pieces of information about a function or an activation record. `lua_getstack` fills only the private part of this structure, for later use. To fill the other fields of `lua_Debug` with useful information, call `lua_getinfo`.

The fields of `lua_Debug` have the following meaning:

- `source`: the source of the chunk that created the function. If `source` starts with a `'@'`, it means that the function was defined in a file where the file name follows the `'@'`. If `source` starts with a `'='`, the remainder of its contents describe the source in a user-dependent manner. Otherwise, the function was defined in a string where `source` is that string.
- `short_src`: a 'printable' version of `source`, to be used in error messages.
- `linedefined`: the line number where the definition of the function starts.
- `lastlinedefined`: the line number where the definition of the function ends.
- `what`: the string `"Lua"` if the function is a Lua function, `"C"` if it is a C function, `"main"` if it is the main part of a chunk.
- `currentline`: the current line where the given function is executing. When no line information is available, `currentline` is set to -1.
- `name`: a reasonable name for the given function. Because functions in Lua are first-class values, they do not have a fixed name: some functions can be the value of multiple global variables, while others can be stored only in a table field. The `lua_getinfo` function checks how the function was called to find a suitable name. If it cannot find a name, then `name` is set to `NULL`.
- `namewhat`: explains the `name` field. The value of `namewhat` can be `"global"`, `"local"`, `"method"`, `"field"`, `"upvalue"`, or `""` (the empty string), according to how the function was called. (Lua uses the empty string when no other option seems to apply.)
- `istailcall`: true if this function invocation was called by a tail call. In this case, the caller of this level is not in the stack.
- `nups`: the number of upvalues of the function.
- `nparams`: the number of fixed parameters of the function (always 0 for C functions).
- `isvararg`: true if the function is a vararg function (always true for C functions).

`lua_gethook` $[-0, +0, -]$

```
lua_Hook lua_gethook (lua_State *L);
```

Returns the current hook function.

`lua_gethookcount` $[-0, +0, -]$

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

Returns the current hook count.

`lua_gethookmask` $[-0, +0, -]$

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

Returns the current hook mask.

`lua_getinfo` $[-(0|1), +(0|1|2), e]$

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

Gets information about a specific function or function invocation.

To get information about a function invocation, the parameter `ar` must be a valid activation record that was filled by a previous call to `lua_getstack` or given as argument to a hook (see `lua_Hook`).

To get information about a function you push it onto the stack and start the `what` string with the character `>`. (In that case, `lua_getinfo` pops the function from the top of the stack.) For instance, to know in which line a function `f` was defined, you can write the following code:

```
lua_Debug ar;
lua_getglobal(L,"f");  /* get global 'f' */
lua_getinfo(L,">S",&ar);
printf("%d\n",ar.linedefined);
```

Each character in the string `what` selects some fields of the structure `ar` to be filled or a value to be pushed on the stack:

- `'n'`: fills in the field `name` and `namewhat`;
- `'S'`: fills in the fields `source`, `short_src`, `linedefined`, `lastlinedefined`, and `what`;
- `'l'`: fills in the field `currentline`;
- `'t'`: fills in the field `istailcall`;
- `'u'`: fills in the fields `nups`, `nparams`, and `isvararg`;
- `'f'`: pushes onto the stack the function that is running at the given level;
- `'L'`: pushes onto the stack a table whose indices are the numbers of the lines that are valid on the function. (A *valid line* is a line with some associated code, that is, a line where you can put a break point. Non-valid lines include empty lines and comments.)

This function returns 0 on error (for instance, an invalid option in `what`).

`lua_getlocal` $[-0, +(0|1), -]$

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

Gets information about a local variable of a given activation record or a given function.

In the first case, the parameter `ar` must be a valid activation record that was filled by a previous call to `lua_getstack` or given as argument to a hook (see `lua_Hook`). The index `n` selects which local variable to inspect; see `debug.getlocal` for details about variable indices and names.

`lua_getlocal` pushes the variable's value onto the stack and returns its name.

In the second case, `ar` should be `NULL` and the function to be inspected must be at the top of the stack. In this case, only parameters of Lua functions are visible (as there is no information about what variables are active) and no values are pushed onto the stack.

Returns `NULL` (and pushes nothing) when the index is greater than the number of active local variables.

`lua_getstack` $[-0, +0, -]$

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

Gets information about the interpreter runtime stack.

This function 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, whereas level $n+1$ is the function that has called level n (except for tail calls, which do not count on the stack). When there are no errors, `lua_getstack` returns 1; when called with a level greater than the stack depth, it returns 0.

`lua_getupvalue` $[-0, +(0|1), -]$

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

Gets information about a closure's upvalue. (For Lua functions, upvalues are the external local variables that the function uses, and that are consequently included in its closure.) `lua_getupvalue` gets the index `n` of an upvalue, pushes the upvalue's value onto the stack, and returns its name. `funcindex` points to the closure in the stack. (Upvalues have no particular order, as they are active through the whole function. So, they are numbered in an arbitrary order.)

Returns `NULL` (and pushes nothing) when the index is greater than the number of upvalues. For C functions, this function uses the empty string `""` as a name for all upvalues.

`lua_Hook`

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

Type for debugging hook functions.

Whenever a hook is called, its `ar` argument has its field `event` set to the specific event that triggered the hook. Lua identifies these events with the following constants: `LUA_HOOKCALL`, `LUA_HOOKRET`, `LUA_HOOKTAILCALL`,

LUA_HOOKLINE, and LUA_HOOKCOUNT. Moreover, for line events, the field currentline is also set. To get the value of any other field in ar, the hook must call lua_getinfo.

For call events, event can be LUA_HOOKCALL, the normal value, or LUA_HOOKTAILCALL, for a tail call; in this case, there will be no corresponding return event.

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, this execution occurs without any calls to hooks.

Hook functions cannot have continuations, that is, they cannot call lua_yieldk, lua_pcallk, or lua_callk with a non-null k.

Hook functions can yield under the following conditions: Only count and line events can yield and they cannot yield any value; to yield a hook function must finish its execution calling lua_yield with nresults equal to zero.

---

lua_sethook $[-0, +0, -]$

```
int lua_sethook (lua_State *L, lua_Hook f,
                 int mask, int count);
```

Sets the debugging hook function.

Argument f is the hook function. mask specifies on which events the hook will be called: it is formed by a bitwise or of the constants LUA_MASKCALL, LUA_MASKRET, LUA_MASKLINE, and LUA_MASKCOUNT. The count argument is only meaningful when the mask includes LUA_MASKCOUNT. For each event, the hook is called as explained below:

The call hook is called when the interpreter calls a function. The hook is called just after Lua enters the new function, before the function gets its arguments.

The return hook is called when the interpreter returns from a function. The hook is called just before Lua leaves the function. There is no standard way to access the values to be returned by the function.

The line hook is called when the interpreter is about to start the execution of a new line of code, or when it jumps back in the code (even to the same line). (This event only happens while Lua is executing a Lua function.)

The count hook is called after the interpreter executes every count instructions. (This event only happens while Lua is executing a Lua function.)

A hook is disabled by setting mask to zero.

---

lua_setlocal $[-(0|1), +0, -]$

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

Sets the value of a local variable of a given activation record. Parameters ar and n are as in lua_getlocal (see lua_getlocal). lua_setlocal assigns the value at the top of the stack to the variable and returns its name. It also pops the value from the stack.

Returns NULL (and pops nothing) when the index is greater than the number of active local variables.

---

lua_setupvalue $[-(0|1), +0, -]$

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

Sets the value of a closure's upvalue. It assigns the value at the top of the stack to the upvalue and returns its name. It also pops the value from the stack. Parameters funcindex and n are as in the lua_getupvalue (see lua_getupvalue).

Returns NULL (and pops nothing) when the index is greater than the number of upvalues.

---

lua_upvalueid $[-0, +0, -]$

```
void *lua_upvalueid
          (lua_State *L, int funcindex, int n);
```

Returns an unique identifier for the upvalue numbered n from the closure at index funcindex. Parameters funcindex and n are as in the lua_getupvalue (see lua_getupvalue) (but n cannot be greater than the number of upvalues).

These unique identifiers allow a program to check whether different closures share upvalues. Lua closures that share an upvalue (that is, that access a same external local variable) will return identical ids for those upvalue indices.

---

lua_upvaluejoin $[-0, +0, -]$

```
void lua_upvaluejoin (lua_State *L,
                      int funcindex1, int n1,
                      int funcindex2, int n2);
```

Make the n1-th upvalue of the Lua closure at index funcindex1 refer to the n2-th upvalue of the Lua closure at index funcindex2.

5 · The Auxiliary Library

The *auxiliary library* provides several convenient functions to interface C with Lua. While the basic API provides the primitive functions for all interactions between C and Lua, the auxiliary library provides higher-level functions for some common tasks.

All functions and types from the auxiliary library are defined in header file lauxlib.h and have a prefix luaL_.

All functions in the auxiliary library are built on top of the basic API, and so they provide nothing that cannot be done with that API. Nevertheless, the use of the auxiliary library ensures more consistency to your code.

Several functions in the auxiliary library use internally some extra stack slots. When a function in the auxiliary library uses less than five slots, it does not check the stack size; it simply assumes that there are enough slots.

Several functions in the auxiliary library are used to check C function arguments. Because the error message

is formatted for arguments (e.g., "`bad argument #1`"), you should not use these functions for other stack values.

Functions called `luaL_check*` always throw an error if the check is not satisfied.

5.1 · Functions and Types

Here we list all functions and types from the auxiliary library in alphabetical order.

`luaL_addchar` $[-?, +?, e]$

```
void luaL_addchar (luaL_Buffer *B, char c);
```

Adds the byte `c` to the buffer `B` (see `luaL_Buffer`).

`luaL_addlstring` $[-?, +?, e]$

```
void luaL_addlstring (luaL_Buffer *B,
                      const char *s, size_t l);
```

Adds the string pointed to by `s` with length `l` to the buffer `B` (see `luaL_Buffer`). The string can contain embedded zeros.

`luaL_addsize` $[-?, +?, e]$

```
void luaL_addsize (luaL_Buffer *B, size_t n);
```

Adds to the buffer `B` (see `luaL_Buffer`) a string of length `n` previously copied to the buffer area (see `luaL_prepbuffer`).

`luaL_addstring` $[-?, +?, e]$

```
void luaL_addstring (luaL_Buffer *B,
                     const char *s);
```

Adds the zero-terminated string pointed to by `s` to the buffer `B` (see `luaL_Buffer`). The string cannot contain embedded zeros.

`luaL_addvalue` $[-1, +?, e]$

```
void luaL_addvalue (luaL_Buffer *B);
```

Adds the value at the top of the stack to the buffer `B` (see `luaL_Buffer`). Pops the value.

This is the only function on string buffers that can (and must) be called with an extra element on the stack, which is the value to be added to the buffer.

`luaL_argcheck` $[-0, +0, v]$

```
void luaL_argcheck (lua_State *L,
                    int cond,
                    int arg,
                    const char *extramsg);
```

Checks whether `cond` is true. If not, raises an error with a standard message.

`luaL_argerror` $[-0, +0, v]$

```
int luaL_argerror (lua_State *L, int arg,
                   const char *extramsg);
```

Raises an error with a standard message that includes `extramsg` as a comment.

This function never returns, but it is an idiom to use it in C functions as `return luaL_argerror(args)`.

`luaL_Buffer`

```
typedef struct luaL_Buffer luaL_Buffer;
```

Type for a *string buffer*.

A string buffer allows C code to build Lua strings piecemeal. Its pattern of use is as follows:

- First declare a variable `b` of type `luaL_Buffer`.
- Then initialize it with a call `luaL_buffinit(L,&b)`.
- Then add string pieces to the buffer calling any of the `luaL_add*` functions.
- Finish by calling `luaL_pushresult(&b)`. This call leaves the final string on the top of the stack.

If you know beforehand the total size of the resulting string, you can use the buffer like this:

- First declare a variable `b` of type `luaL_Buffer`.
- Then initialize it and preallocate a space of size `sz` with a call `luaL_buffinitsize(L,&b,sz)`.
- Then copy the string into that space.
- Finish by calling `luaL_pushresultsize(&b,sz)`, where `sz` is the total size of the resulting string copied into that space.

During its normal operation, a string buffer uses a variable number of stack slots. So, while using a buffer, you cannot assume that you know where the top of the stack is. You can use the stack between successive calls to buffer operations as long as that use is balanced; that is, when you call a buffer operation, the stack is at the same level it was immediately after the previous buffer operation. (The only exception to this rule is `luaL_addvalue`.) After calling `luaL_pushresult` the stack is back to its level when the buffer was initialized, plus the final string on its top.

`luaL_buffinit` $[-0, +0, -]$

```
void luaL_buffinit (lua_State *L,
                    luaL_Buffer *B);
```

Initializes a buffer `B`. This function does not allocate any space; the buffer must be declared as a variable (see `luaL_Buffer`).

`luaL_buffinitsize` $[-?, +?, e]$

```
char *luaL_buffinitsize
     (lua_State *L, luaL_Buffer *B, size_t sz);
```

Equivalent to the sequence `luaL_buffinit`, `luaL_prepbuffsize`.

`luaL_callmeta` $[-0, +(0|1), e]$

```
int luaL_callmeta (lua_State *L, int obj,
                   const char *e);
```

Calls a metamethod.

If the object at index `obj` has a metatable and this metatable has a field `e`, this function calls this field passing the object as its only argument. In this case this function returns true and pushes onto the stack the value returned by the call. If there is no metatable or no metamethod, this function returns false (without pushing any value on the stack).

`luaL_checkany` $[-0, +0, v]$

```
void luaL_checkany (lua_State *L, int arg);
```

Checks whether the function has an argument of any type (including `nil`) at position `arg`.

`luaL_checkint` $[-0, +0, v]$

```
int luaL_checkint (lua_State *L, int arg);
```

Checks whether the function argument `arg` is a number and returns this number cast to an `int`.

`luaL_checkinteger` $[-0, +0, v]$

```
lua_Integer luaL_checkinteger (lua_State *L,
                               int arg);
```

Checks whether the function argument `arg` is a number and returns this number cast to a `lua_Integer`.

`luaL_checklong` $[-0, +0, v]$

```
long luaL_checklong (lua_State *L, int arg);
```

Checks whether the function argument `arg` is a number and returns this number cast to a `long`.

`luaL_checklstring` $[-0, +0, v]$

```
const char *luaL_checklstring
            (lua_State *L, int arg, size_t *l);
```

Checks whether the function argument `arg` is a string and returns this string; if `l` is not `NULL` fills `*l` with the string's length.

This function uses `lua_tolstring` to get its result, so all conversions and caveats of that function apply here.

`luaL_checknumber` $[-0, +0, v]$

```
lua_Number luaL_checknumber (lua_State *L,
                             int arg);
```

Checks whether the function argument `arg` is a number and returns this number.

`luaL_checkoption` $[-0, +0, v]$

```
int luaL_checkoption (lua_State *L,
                      int arg,
                      const char *def,
                      const char *const lst[]);
```

Checks whether the function argument `arg` is a string and searches for this string in the array `lst` (which must be NULL-terminated). Returns the index in the array where the string was found. Raises an error if the argument is not a string or if the string cannot be found.

If `def` is not `NULL`, the function uses `def` as a default value when there is no argument `arg` or when this argument is `nil`.

This is a useful function for mapping strings to C enums. (The usual convention in Lua libraries is to use strings instead of numbers to select options.)

`luaL_checkstack` $[-0, +0, v]$

```
void luaL_checkstack (lua_State *L, int sz,
                      const char *msg);
```

Grows the stack size to `top + sz` elements, raising an error if the stack cannot grow to that size. `msg` is an additional text to go into the error message (or `NULL` for no additional text).

`luaL_checkstring` $[-0, +0, v]$

```
const char *luaL_checkstring (lua_State *L,
                              int arg);
```

Checks whether the function argument `arg` is a string and returns this string.

This function uses `lua_tolstring` to get its result, so all conversions and caveats of that function apply here.

`luaL_checktype` $[-0, +0, v]$

```
void luaL_checktype (lua_State *L, int arg,
                     int t);
```

Checks whether the function argument `arg` has type `t`. See `lua_type` for the encoding of types for `t`.

`luaL_checkudata` $[-0, +0, v]$

```
void *luaL_checkudata (lua_State *L, int arg,
                       const char *tname);
```

Checks whether the function argument `arg` is a userdata of the type `tname` (see `luaL_newmetatable`) and returns the userdata address (see `lua_touserdata`).

`luaL_checkunsigned` $[-0, +0, v]$

```
lua_Unsigned luaL_checkunsigned (lua_State *L,
                                 int arg);
```

Checks whether the function argument `arg` is a number and returns this number cast to a `lua_Unsigned`.

`luaL_checkversion` $[-0, +0, -]$

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

Checks whether the core running the call, the core that created the Lua state, and the code making the call are all using the same version of Lua. Also checks whether the core running the call and the core that created the Lua state are using the same address space.

`luaL_dofile` $[-0, +?, e]$

```
int luaL_dofile (lua_State *L,
                 const char *filename);
```

Loads and runs the given file. It is defined as the following macro:

```
(luaL_loadfile(L, filename) ||
 lua_pcall(L, 0, LUA_MULTRET, 0))
```

It returns false if there are no errors or true in case of errors.

`luaL_dostring` $[-0, +?, -]$

```
int luaL_dostring (lua_State *L,
                   const char *str);
```

Loads and runs the given string. It is defined as the following macro:

```
(luaL_loadstring(L, str) ||
 lua_pcall(L, 0, LUA_MULTRET, 0))
```

It returns false if there are no errors or true in case of errors.

`luaL_error` $[-0, +0, v]$

```
int luaL_error (lua_State *L,
                const char *fmt, ...);
```

Raises an error. The error message format is given by `fmt` plus any extra arguments, following the same rules of `lua_pushfstring`. It also adds at the beginning of the message the file name and the line number where the error occurred, if this information is available.

This function never returns, but it is an idiom to use it in C functions as `return luaL_error(args)`.

`luaL_execresult` $[-0, +3, e]$

```
int luaL_execresult (lua_State *L, int stat);
```

This function produces the return values for process-related functions in the standard library (`os.execute` and `io.close`).

`luaL_fileresult` $[-0, +(1|3), e]$

```
int luaL_fileresult (lua_State *L, int stat,
                     const char *fname);
```

This function produces the return values for file-related functions in the standard library (`io.open`, `os.rename`, `file:seek`, etc.).

`luaL_getmetafield` $[-0, +(0|1), e]$

```
int luaL_getmetafield (lua_State *L, int obj,
                       const char *e);
```

Pushes onto the stack the field `e` from the metatable of the object at index `obj`. If the object does not have a metatable, or if the metatable does not have this field, returns false and pushes nothing.

`luaL_getmetatable` $[-0, +1, -]$

```
void luaL_getmetatable (lua_State *L,
                        const char *tname);
```

Pushes onto the stack the metatable associated with name `tname` in the registry (see `luaL_newmetatable`).

`luaL_getsubtable` $[-0, +1, e]$

```
int luaL_getsubtable (lua_State *L, int idx,
                      const char *fname);
```

Ensures that the value `t[fname]`, where `t` is the value at the valid index `idx`, is a table, and pushes that table onto the stack. Returns true if it finds a previous table there and false if it creates a new table.

`luaL_gsub` $[-0, +1, e]$

```
const char *luaL_gsub (lua_State *L,
                       const char *s,
                       const char *p,
                       const char *r);
```

Creates a copy of string `s` by replacing any occurrence of the string `p` with the string `r`. Pushes the resulting string on the stack and returns it.

`luaL_len` $[-0, +0, e]$

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

Returns the 'length' of the value at the given acceptable index as a number; it is equivalent to the `#` operator in Lua (see §3.4.6). Raises an error if the result of the operation is not a number. (This case only can happen through metamethods.)

`luaL_loadbuffer` $[-0, +1, -]$

```
int luaL_loadbuffer (lua_State *L,
                     const char *buff,
                     size_t sz,
                     const char *name);
```

Equivalent to `luaL_loadbufferx` with `mode` equal to `NULL`.

`luaL_loadbufferx` $[-0, +1, -]$

```
int luaL_loadbufferx (lua_State *L,
                      const char *buff,
                      size_t sz,
                      const char *name,
                      const char *mode);
```

Loads a buffer as a Lua chunk. This function uses `lua_load` to load the chunk in the buffer pointed to by `buff` with size `sz`.

This function returns the same results as `lua_load`. `name` is the chunk name, used for debug information and error messages. The string `mode` works as in function `lua_load`.

`luaL_loadfile` $[-0, +1, e]$

```
int luaL_loadfile (lua_State *L,
                   const char *filename);
```

Equivalent to `luaL_loadfilex` with `mode` equal to `NULL`.

`luaL_loadfilex` $[-0, +1, e]$

```
int luaL_loadfilex (lua_State *L,
                    const char *filename,
                    const char *mode);
```

Loads a file as a Lua chunk. This function uses `lua_load` to load the chunk in the file named `filename`. If `filename` is `NULL`, then it loads from the standard input. The first line in the file is ignored if it starts with a `#`.

The string `mode` works as in function `lua_load`.

This function returns the same results as `lua_load`, but it has an extra error code `LUA_ERRFILE` if it cannot open/read the file or the file has a wrong mode.

As `lua_load`, this function only loads the chunk; it does not run it.

`luaL_loadstring` $[-0, +1, -]$

```
int luaL_loadstring (lua_State *L,
                     const char *s);
```

Loads a string as a Lua chunk. Uses `lua_load` to load the chunk in the zero-terminated string `s`.

This function returns the same results as `lua_load`.

Also as `lua_load`, this function only loads the chunk; it does not run it.

`luaL_newlib` $[-0, +1, e]$

```
int luaL_newlib (lua_State *L,
                 const luaL_Reg *l);
```

Creates a new table and registers there the functions in list `l`. It is implemented as the following macro:

```
(luaL_newlibtable(L,l), luaL_setfuncs(L,l,0))
```

`luaL_newlibtable` $[-0, +1, e]$

```
int luaL_newlibtable (lua_State *L,
                      const luaL_Reg l[]);
```

Creates a new table with a size optimized to store all entries in the array `l` (but does not actually store them). It is intended to be used in conjunction with `luaL_setfuncs` (see `luaL_newlib`).

It is implemented as a macro. The array `l` must be the actual array, not a pointer to it.

`luaL_newmetatable` $[-0, +1, e]$

```
int luaL_newmetatable (lua_State *L,
                       const char *tname);
```

If the registry already has the key `tname`, returns 0. Otherwise, creates a new table to be used as a metatable for userdata, adds it to the registry with key `tname`, and returns 1.

In both cases pushes onto the stack the final value associated with `tname` in the registry.

`luaL_newstate` $[-0, +0, -]$

```
lua_State *luaL_newstate (void);
```

Creates a new Lua state. It calls `lua_newstate` with an allocator based on the standard C `realloc` function and then sets a panic function (see §4.6) that prints an error message to the standard error output in case of fatal errors.

Returns the new state, or `NULL` if there is a memory allocation error.

`luaL_openlibs` $[-0, +0, e]$

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

Opens all standard Lua libraries into the given state.

`luaL_optint` $[-0, +0, v]$

```
int luaL_optint (lua_State *L, int arg, int d);
```

If the function argument `arg` is a number, returns this number cast to an `int`. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

`luaL_optinteger` $[-0, +0, v]$

```
lua_Integer luaL_optinteger (lua_State *L,
                             int arg,
                             lua_Integer d);
```

If the function argument `arg` is a number, returns this number cast to a `lua_Integer`. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

`luaL_optlong` $[-0, +0, v]$

```
long luaL_optlong (lua_State *L, int arg,
                   long d);
```

If the function argument `arg` is a number, returns this number cast to a `long`. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

`luaL_optlstring` $[-0, +0, v]$

```
const char *luaL_optlstring (lua_State *L,
                             int arg,
                             const char *d,
                             size_t *l);
```

If the function argument `arg` is a string, returns this string. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

If `l` is not `NULL`, fills the position `*l` with the result's length.

`luaL_optnumber` $[-0, +0, v]$

```
lua_Number luaL_optnumber
         (lua_State *L, int arg, lua_Number d);
```

If the function argument `arg` is a number, returns this number. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

`luaL_optstring` $[-0, +0, v]$

```
const char *luaL_optstring (lua_State *L,
                            int arg,
                            const char *d);
```

If the function argument `arg` is a string, returns this string. If this argument is absent or is `nil`, returns `d`. Otherwise, raises an error.

`luaL_optunsigned` $[-0, +0, v]$

```
lua_Unsigned luaL_optunsigned (lua_State *L,
                               int arg,
                               lua_Unsigned u);
```

If the function argument `arg` is a number, returns this number cast to a `lua_Unsigned`. If this argument is absent or is `nil`, returns `u`. Otherwise, raises an error.

`luaL_prepbuffer` $[-?, +?, e]$

```
char *luaL_prepbuffer (luaL_Buffer *B);
```

Equivalent to `luaL_prepbuffsize` with the predefined size `LUAL_BUFFERSIZE`.

`luaL_prepbuffsize` $[-?, +?, e]$

```
char *luaL_prepbuffsize (luaL_Buffer *B,
                         size_t sz);
```

Returns an address to a space of size `sz` where you can copy a string to be added to buffer `B` (see `luaL_Buffer`). After copying the string into this space you must call `luaL_addsize` with the size of the string to actually add it to the buffer.

`luaL_pushresult` $[-?, +1, e]$

```
void luaL_pushresult (luaL_Buffer *B);
```

Finishes the use of buffer `B` leaving the final string on the top of the stack.

`luaL_pushresultsize` $[-?, +1, e]$

```
void luaL_pushresultsize (luaL_Buffer *B,
                          size_t sz);
```

Equivalent to the sequence `luaL_addsize`, `luaL_pushresult`.

`luaL_ref` $[-1, +0, e]$

```
int luaL_ref (lua_State *L, int t);
```

Creates and returns a *reference*, in the table at index `t`, for the object at the top of the stack (and pops the object).

A reference is a unique integer key. As long as you do not manually add integer keys into table `t`, `luaL_ref` ensures the uniqueness of the key it returns. You can retrieve an object referred by reference `r` by calling `lua_rawgeti(L, t, r)`. Function `luaL_unref` frees a reference and its associated object.

If the object at the top of the stack is `nil`, `luaL_ref` returns the constant `LUA_REFNIL`. The constant `LUA_NOREF` is guaranteed to be different from any reference returned by `luaL_ref`.

`luaL_Reg`

```
typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;
```

Type for arrays of functions to be registered by `luaL_setfuncs`. `name` is the function name and `func` is a pointer to the function. Any array of `luaL_Reg` must end with an sentinel entry in which both `name` and `func` are `NULL`.

`luaL_requiref` $[-0, +1, e]$

```
void luaL_requiref
            (lua_State *L, const char *modname,
             lua_CFunction openf, int glb);
```

Calls function `openf` with string `modname` as an argument and sets the call result in `package.loaded[modname]`, as if that function has been called through `require`.

If `glb` is true, also stores the result into global `modname`.

Leaves a copy of that result on the stack.

`luaL_setfuncs` $[-nup, +0, e]$

```
void luaL_setfuncs
    (lua_State *L, const luaL_Reg *l, int nup);
```

Registers all functions in the array `l` (see `luaL_Reg`) into the table on the top of the stack (below optional upvalues, see next).

When `nup` is not zero, all functions are created sharing `nup` upvalues, which must be previously pushed on the stack on top of the library table. These values are popped from the stack after the registration.

`luaL_setmetatable` $[-0, +0, -]$

```
void luaL_setmetatable (lua_State *L,
                        const char *tname);
```

Sets the metatable of the object at the top of the stack as the metatable associated with name `tname` in the registry (see `luaL_newmetatable`).

`luaL_testudata` $[-0, +0, e]$

```
void *luaL_testudata (lua_State *L, int arg,
                      const char *tname);
```

This function works like `luaL_checkudata`, except that, when the test fails, it returns `NULL` instead of throwing an error.

`luaL_tolstring` $[-0, +1, e]$

```
const char *luaL_tolstring
          (lua_State *L, int idx, size_t *len);
```

Converts any Lua value at the given acceptable index to a C string in a reasonable format. The resulting string is pushed onto the stack and also returned by the function. If `len` is not `NULL`, the function also sets `*len` with the string length.

If the value has a metatable with a `"__tostring"` field, then `luaL_tolstring` calls the corresponding metamethod with the value as argument, and uses the result of the call as its result.

`luaL_traceback` $[-0, +1, e]$

```
void luaL_traceback
                  (lua_State *L, lua_State *L1,
                   const char *msg, int level);
```

Creates and pushes a traceback of the stack `L1`. If `msg` is not `NULL` it is appended at the beginning of the traceback. The `level` parameter tells at which level to start the traceback.

`luaL_typename` $[-0, +0, -]$

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

Returns the name of the type of the value at the given index.

`luaL_unref` $[-0, +0, -]$

```
void luaL_unref (lua_State *L, int t, int ref);
```

Releases reference `ref` from the table at index `t` (see `luaL_ref`). The entry is removed from the table, so that the referred object can be collected. The reference `ref` is also freed to be used again.

If `ref` is `LUA_NOREF` or `LUA_REFNIL`, `luaL_unref` does nothing.

`luaL_where` $[-0, +1, e]$

```
void luaL_where (lua_State *L, int lvl);
```

Pushes onto the stack a string identifying the current position of the control at level `lvl` in the call stack. Typically this string has the following format:

```
chunkname:currentline:
```

Level 0 is the running function, level 1 is the function that called the running function, etc.

This function is used to build a prefix for error messages.

6 · Standard Libraries

The standard Lua libraries provide useful functions that are implemented directly through the C API. Some of these functions provide essential services to the language (e.g., `type` and `getmetatable`); others provide access to "outside" services (e.g., I/O); and others could be implemented in Lua itself, but are quite useful or have critical performance requirements that deserve an implementation in C (e.g., `table.sort`).

All libraries are implemented through the official C API and are provided as separate C modules. Currently, Lua has the following standard libraries:

- basic library (§6.1);
- coroutine library (§6.2);
- package library (§6.3);
- string manipulation (§6.4);
- table manipulation (§6.5);
- mathematical functions (§6.6) (sin, log, etc.);
- bitwise operations (§6.7);
- input and output (§6.8);
- operating system facilities (§6.9);
- debug facilities (§6.10).

Except for the basic and the package libraries, each library provides all its functions as fields of a global table or as methods of its objects.

To have access to these libraries, the C host program should call the `luaL_openlibs` function, which opens all standard libraries. Alternatively, the host program can open them individually by using `luaL_requiref` to call `luaopen_base` (for the basic library), `luaopen_package` (for the package library), `luaopen_coroutine` (for the coroutine library), `luaopen_string` (for the string library), `luaopen_table` (for the table library), `luaopen_math` (for the mathematical library), `luaopen_bit32` (for the bit library), `luaopen_io` (for the I/O library), `luaopen_os` (for the Operating System library), and `luaopen_debug` (for the debug library). These functions are declared in `lualib.h`.

6.1 · Basic Functions

The basic library provides core functions to Lua. If you do not include this library in your application, you should check carefully whether you need to provide implementations for some of its facilities.

`assert (v [, message])`

Issues an error when the value of its argument `v` is false (i.e., `nil` or `false`); otherwise, returns all its arguments. `message` is an error message; when absent, it defaults to "assertion failed!"

`collectgarbage ([opt [, arg]])`

This function is a generic interface to the garbage collector. It performs different functions according to its first argument, `opt`:

- `"collect"`: performs a full garbage-collection cycle. This is the default option.
- `"stop"`: stops automatic execution of the garbage collector. The collector will run only when explicitly invoked, until a call to restart it.
- `"restart"`: restarts automatic execution of the garbage collector.
- `"count"`: returns the total memory in use by Lua (in Kbytes) and a second value with the total memory in bytes modulo 1024. The first value has a fractional part, so the following equality is always true:
  ```
  k,b = collectgarbage("count")
  assert(k*1024 == math.floor(k)*1024+b)
  ```
  (The second result is useful when Lua is compiled with a non floating-point type for numbers.)
- `"step"`: performs a garbage-collection step. The step "size" is controlled by `arg` (larger values mean more steps) in a non-specified way. If you want to control the step size you must experimentally tune the value of `arg`. Returns `true` if the step finished a collection cycle.
- `"setpause"`: sets `arg` as the new value for the *pause* of the collector (see §2.5). Returns the previous value for *pause*.
- `"setstepmul"`: sets `arg` as the new value for the *step multiplier* of the collector (see §2.5). Returns the previous value for *step*.
- `"isrunning"`: returns a boolean that tells whether the collector is running (i.e., not stopped).
- `"generational"`: changes the collector to generational mode. This is an experimental feature (see §2.5).
- `"incremental"`: changes the collector to incremental mode. This is the default mode.

`dofile ([filename])`

Opens the named file and executes its contents as a Lua chunk. When called without arguments, `dofile` executes the contents of the standard input (`stdin`). Returns all values returned by the chunk. In case of errors, `dofile` propagates the error to its caller (that is, `dofile` does not run in protected mode).

`error (message [, level])`

Terminates the last protected function called and returns `message` as the error message. Function `error` never returns.

Usually, `error` adds some information about the error position at the beginning of the message, if the message is a string. The `level` argument specifies how to get the error position. With level 1 (the default), the error position is where the `error` function was called. Level 2 points the error to where the function that called `error` was called; and so on. Passing a level 0 avoids the addition of error position information to the message.

`_G`

A global variable (not a function) that holds the global environment (see §2.2). Lua itself does not use this variable; changing its value does not affect any environment, nor vice-versa.

`getmetatable (object)`

If `object` does not have a metatable, returns `nil`. Otherwise, if the object's metatable has a `"__metatable"` field, returns the associated value. Otherwise, returns the metatable of the given object.

`ipairs (t)`

If `t` has a metamethod `__ipairs`, calls it with `t` as argument and returns the first three results from the call.

Otherwise, returns three values: an iterator function, the table `t`, and 0, so that the construction

```
for i,v in ipairs(t) do body end
```

will iterate over the pairs `(1,t[1])`, `(2,t[2])`, ..., up to the first integer key absent from the table.

`load (ld [, source [, mode [, env]]])`

Loads a chunk.

If `ld` is a string, the chunk is this string. If `ld` is a function, `load` calls it repeatedly to get the chunk pieces. Each call to `ld` must return a string that concatenates with previous results. A return of an empty string, `nil`, or no value signals the end of the chunk.

If there are no syntactic errors, returns the compiled chunk as a function; otherwise, returns `nil` plus the error message.

If the resulting function has upvalues, the first upvalue is set to the value of the global environment or to `env`, if that parameter is given. When loading main chunks, the first upvalue will be the `_ENV` variable (see §2.2).

`source` is used as the source of the chunk for error messages and debug information (see §4.9). When absent, it defaults to `ld`, if `ld` is a string, or to `=(load)` otherwise.

The string `mode` controls whether the chunk can be text or binary (that is, a precompiled chunk). It may be the string `"b"` (only binary chunks), `"t"` (only text chunks), or `"bt"` (both binary and text). The default is `"bt"`.

`loadfile ([filename [, mode [, env]]])`

Similar to `load`, but gets the chunk from file `filename` or from the standard input, if no file name is given.

`next (table [, index])`

Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. `next` returns the next index of the table and its associated value. When called with `nil` as its second argument, `next` returns an initial index and its associated value. When called with the last index, or with `nil` in an empty table, `next` returns `nil`. If the second argument is absent, then it is interpreted as `nil`. In particular, you can use `next(t)` to check whether a table is empty.

The order in which the indices are enumerated is not specified, *even for numeric indices*. (To traverse a table in numeric order, use a numerical `for`.)

The behavior of `next` is undefined if, during the traversal, you assign any value to a non-existent field in the table. You may however modify existing fields. In particular, you may clear existing fields.

`pairs (t)`

If `t` has a metamethod `__pairs`, calls it with `t` as argument and returns the first three results from the call.

Otherwise, returns three values: the `next` function, the table `t`, and `nil`, so that the construction

```
for k,v in pairs(t) do body end
```

will iterate over all key—value pairs of table `t`.

See function `next` for the caveats of modifying the table during its traversal.

`pcall (f [, arg1, ···])`

Calls function `f` with the given arguments in *protected mode*. This means that any error inside `f` is not propagated; instead, `pcall` catches the error and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, `pcall` also returns all results from the call, after this first result. In case of any error, `pcall` returns `false` plus the error message.

`print (···)`

Receives any number of arguments and prints their values to `stdout`, using the `tostring` function to convert each argument to a string. `print` is not intended for formatted output, but only as a quick way to show a value, for instance for debugging. For complete control over the output, use `string.format` and `io.write`.

`rawequal (v1, v2)`

Checks whether `v1` is equal to `v2`, without invoking any metamethod. Returns a boolean.

`rawget (table, index)`

Gets the real value of `table[index]`, without invoking any metamethod. `table` must be a table; `index` may be any value.

`rawlen (v)`

Returns the length of the object `v`, which must be a table or a string, without invoking any metamethod. Returns an integer number.

`rawset (table, index, value)`

Sets the real value of `table[index]` to `value`, without invoking any metamethod. `table` must be a table, `index` any value different from `nil` and NaN, and `value` any Lua value.

This function returns `table`.

`select (index, ···)`

If `index` is a number, returns all arguments after argument number `index`; a negative number indexes from the end (-1 is the last argument). Otherwise, `index` must be the string `"#"`, and `select` returns the total number of extra arguments it received.

`setmetatable (table, metatable)`

Sets the metatable for the given table. (You cannot change the metatable of other types from Lua, only from C.) If `metatable` is `nil`, removes the metatable of the given table. If the original metatable has a `"__metatable"` field, raises an error.

This function returns `table`.

`tonumber (e [, base])`

When called with no `base`, `tonumber` tries to convert its argument to a number. If the argument is already a number or a string convertible to a number (see §3.4.2),

then `tonumber` returns this number; otherwise, it returns `nil`.

When called with `base`, then `e` should be a string to be interpreted as an integer numeral in that base. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter '`A`' (in either upper or lower case) represents 10, '`B`' represents 11, and so forth, with '`Z`' representing 35. If the string `e` is not a valid numeral in the given base, the function returns `nil`.

`tostring (v)`

Receives a value of any type and converts it to a string in a reasonable format. (For complete control of how numbers are converted, use `string.format`.)

If the metatable of `v` has a `"__tostring"` field, then `tostring` calls the corresponding value with `v` as argument, and uses the result of the call as its result.

`type (v)`

Returns the type of its only argument, coded as a string. The possible results of this function are `"nil"` (a string, not the value `nil`), `"number"`, `"string"`, `"boolean"`, `"table"`, `"function"`, `"thread"`, and `"userdata"`.

`_VERSION`

A global variable (not a function) that holds a string containing the current interpreter version. The current contents of this variable is `"Lua 5.2"`.

`xpcall (f, msgh [, arg1, ···])`

This function is similar to `pcall`, except that it sets a new message handler `msgh`.

6.2 · Coroutine Manipulation

The operations related to coroutines comprise a sublibrary of the basic library and come inside the table `coroutine`. See §2.6 for a general description of coroutines.

`coroutine.create (f)`

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns this new coroutine, an object with type `"thread"`.

`coroutine.resume (co [, val1, ···])`

Starts or continues the execution of coroutine `co`. The first time you resume a coroutine, it starts running its body. The values `val1`, ... are passed as the arguments to the body function. If the coroutine has yielded, `resume` restarts it; the values `val1`, ... are passed as the results from the yield.

If the coroutine runs without any errors, `resume` returns `true` plus any values passed to `yield` (if the coroutine yields) or any values returned by the body function (if the coroutine terminates). If there is any error, `resume` returns `false` plus the error message.

`coroutine.running ()`

Returns the running coroutine plus a boolean, true when the running coroutine is the main one.

`coroutine.status (co)`

Returns the status of coroutine `co`, as a string: `"running"`, if the coroutine is running (that is, it called `status`); `"suspended"`, if the coroutine is suspended in a call to `yield`, or if it has not started running yet; `"normal"` if the coroutine is active but not running (that is, it has resumed another coroutine); and `"dead"` if the coroutine has finished its body function, or if it has stopped with an error.

`coroutine.wrap (f)`

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns a function that resumes the coroutine each time it is called. Any arguments passed to the function behave as the extra arguments to `resume`. Returns the same values returned by `resume`, except the first boolean. In case of error, propagates the error.

`coroutine.yield (···)`

Suspends the execution of the calling coroutine. Any arguments to `yield` are passed as extra results to `resume`.

6.3 · Modules

The package library provides basic facilities for loading modules in Lua. It exports one function directly in the global environment: `require`. Everything else is exported in a table `package`.

`require (modname)`

Loads the given module. The function starts by looking into the `package.loaded` table to determine whether `modname` is already loaded. If it is, then `require` returns the value stored at `package.loaded[modname]`. Otherwise, it tries to find a *loader* for the module.

To find a loader, `require` is guided by the `package.searchers` sequence. By changing this sequence, we can change how `require` looks for a module. The following explanation is based on the default configuration for `package.searchers`.

First `require` queries `package.preload[modname]`. If it has a value, this value (which should be a function) is the loader. Otherwise `require` searches for a Lua loader using the path stored in `package.path`. If that also fails, it searches for a C loader using the path stored in `package.cpath`. If that also fails, it tries an *all-in-one* loader (see `package.searchers`).

Once a loader is found, `require` calls the loader with two arguments: `modname` and an extra value dependent on how it got the loader. (If the loader came from a file, this extra value is the file name.) If the loader returns any non-nil value, `require` assigns the returned value to `package.loaded[modname]`. If the loader does not

return a non-nil value and has not assigned any value to `package.loaded[modname]`, then `require` assigns `true` to this entry. In any case, `require` returns the final value of `package.loaded[modname]`.

If there is any error loading or running the module, or if it cannot find any loader for the module, then `require` raises an error.

`package.config`

A string describing some compile-time configurations for packages. This string is a sequence of lines:

- The first line is the directory separator string. Default is `'\'` for Windows and `'/'` for all other systems.
- The second line is the character that separates templates in a path. Default is `';'`.
- The third line is the string that marks the substitution points in a template. Default is `'?'`.
- The fourth line is a string that, in a path in Windows, is replaced by the executable's directory. Default is `'!'`.
- The fifth line is a mark to ignore all text before it when building the `luaopen_` function name. Default is `'-'`.

`package.cpath`

The path used by `require` to search for a C loader.

Lua initializes the C path `package.cpath` in the same way it initializes the Lua path `package.path`, using the environment variable `LUA_CPATH_5_2` or the environment variable `LUA_CPATH` or a default path defined in `luaconf.h`.

`package.loaded`

A table used by `require` to control which modules are already loaded. When you require a module `modname` and `package.loaded[modname]` is not false, `require` simply returns the value stored there.

This variable is only a reference to the real table; assignments to this variable do not change the table used by `require`.

`package.loadlib (libname, funcname)`

Dynamically links the host program with the C library `libname`.

If `funcname` is `"*"`, then it only links with the library, making the symbols exported by the library available to other dynamically linked libraries. Otherwise, it looks for a function `funcname` inside the library and returns this function as a C function. (So, `funcname` must follow the prototype `lua_CFunction`).

This is a low-level function. It completely bypasses the package and module system. Unlike `require`, it does not perform any path searching and does not automatically adds extensions. `libname` must be the complete file name of the C library, including if necessary a path and an extension. `funcname` must be the exact name exported by the C library (which may depend on the C compiler and linker used).

This function is not supported by Standard C. As such, it is only available on some platforms (Windows, Linux, Mac OS X, Solaris, BSD, plus other Unix systems that support the `dlfcn` standard).

`package.path`

The path used by `require` to search for a Lua loader.

At start-up, Lua initializes this variable with the value of the environment variable `LUA_PATH_5_2` or the environment variable `LUA_PATH` or with a default path defined in `luaconf.h`, if those environment variables are not defined. Any `;;` in the value of the environment variable is replaced by the default path.

`package.preload`

A table to store loaders for specific modules (see `require`).

This variable is only a reference to the real table; assignments to this variable do not change the table used by `require`.

`package.searchers`

A table used by `require` to control how to load modules.

Each entry in this table is a *searcher function*. When looking for a module, `require` calls each of these searchers in ascending order, with the module name (the argument given to `require`) as its sole parameter. The function can return another function (the module *loader*) plus an extra value that will be passed to that loader, or a string explaining why it did not find that module (or `nil` if it has nothing to say).

Lua initializes this table with four searcher functions.

The first searcher simply looks for a loader in the `package.preload` table.

The second searcher looks for a loader as a Lua library, using the path stored at `package.path`. The search is done as described in function `package.searchpath`.

The third searcher looks for a loader as a C library, using the path given by the variable `package.cpath`. Again, the search is done as described in function `package.searchpath`. For instance, if the C path is the string

```
"./?.so;./?.dll;/usr/local/?/init.so"
```

the searcher for module `foo` will try to open the files `./foo.so`, `./foo.dll`, and `/usr/local/foo/init.so`, in that order. Once it finds a C library, this searcher first uses a dynamic link facility to link the application with the library. Then it tries to find a C function inside the library to be used as the loader. The name of this C function is the string `"luaopen_"` concatenated with a copy of the module name where each dot is replaced by an underscore. Moreover, if the module name has a hyphen, its prefix up to (and including) the first hyphen is removed. For instance, if the module name is `a.v1-b.c`, the function name will be `luaopen_b_c`.

The fourth searcher tries an *all-in-one loader*. It searches the C path for a library for the root name of the given module. For instance, when requiring `a.b.c`,

it will search for a C library for `a`. If found, it looks into it for an open function for the submodule; in our example, that would be `luaopen_a_b_c`. With this facility, a package can pack several C submodules into one single library, with each submodule keeping its original open function.

All searchers except the first one (preload) return as the extra value the file name where the module was found, as returned by `package.searchpath`. The first searcher returns no extra value.

`package.searchpath (name,path[,sep[,rep]])`

Searches for the given `name` in the given `path`.

A path is a string containing a sequence of *templates* separated by semicolons. For each template, the function replaces each interrogation mark (if any) in the template with a copy of `name` wherein all occurrences of `sep` (a dot, by default) were replaced by `rep` (the system's directory separator, by default), and then tries to open the resulting file name.

For instance, if the path is the string

```
"./?.lua;./?.lc;/usr/local/?/init.lua"
```

the search for the name `foo.a` will try to open the files `./foo/a.lua`, `./foo/a.lc`, and `/usr/local/foo/a/init.lua`, in that order.

Returns the resulting name of the first file that it can open in read mode (after closing the file), or `nil` plus an error message if none succeeds. (This error message lists all file names it tried to open.)

6.4 · String Manipulation

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. When indexing a string in Lua, the first character is at position 1 (not at 0, as in C). Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position -1, and so on.

The string library provides all its functions inside the table `string`. It also sets a metatable for strings where the `__index` field points to the `string` table. Therefore, you can use the string functions in object-oriented style. For instance, `string.byte(s,i)` can be written as `s:byte(i)`.

The string library assumes one-byte character encodings.

`string.byte (s [, i [, j]])`

Returns the internal numerical codes of the characters `s[i]`, `s[i+1]`, ..., `s[j]`. The default value for `i` is 1; the default value for `j` is `i`. These indices are corrected following the same rules of function `string.sub`.

Numerical codes are not necessarily portable across platforms.

`string.char (···)`

Receives zero or more integers. Returns a string with length equal to the number of arguments, in which each character has the internal numerical code equal to its corresponding argument.

Numerical codes are not necessarily portable across platforms.

`string.dump (function)`

Returns a string containing a binary representation of the given function, so that a later `load` on this string returns a copy of the function (but with new upvalues).

`string.find (s, pattern [, init [, plain]])`

Looks for the first match of `pattern` in the string `s`. If it finds a match, then `find` returns the indices of `s` where this occurrence starts and ends; otherwise, it returns `nil`. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and can be negative. A value of `true` as a fourth, optional argument `plain` turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in `pattern` being considered magic. Note that if `plain` is given, then `init` must be given as well.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

`string.format (formatstring, ···)`

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the C function `sprintf`. The only differences are that the options/modifiers `*`, `h`, `L`, `l`, `n`, and `p` are not supported and that there is an extra option, `q`. The `q` option formats a string between double quotes, using escape sequences when necessary to ensure that it can safely be read back by the Lua interpreter. For instance, the call

```
string.format('%q', 'a string with "quotes" \
                     and \n new line')
```

may produce the string:

```
"a string with \"quotes\" and \
 new line"
```

Options `A` and `a` (when available), `E`, `e`, `f`, `G`, and `g` all expect a number as argument. Options `c`, `d`, `i`, `o`, `u`, `X`, and `x` also expect a number, but the range of that number may be limited by the underlying C implementation. For options `o`, `u`, `X`, and `x`, the number cannot be negative. Option `q` expects a string; option `s` expects a string without embedded zeros. If the argument to option `s` is not a string, it is converted to one following the same rules of `tostring`.

`string.gmatch (s, pattern)`

Returns an iterator function that, each time it is called, returns the next captures from `pattern` over the string `s`. If `pattern` specifies no captures, then the whole match is produced in each call.

As an example, the following loop will iterate over all the words from string `s`, printing one per line:

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
  print(w)
end
```

The next example collects all pairs `key=value` from the given string into a table:

```
t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

For this function, a caret '^' at the start of a pattern does not work as an anchor, as this would prevent the iteration.

`string.gsub (s, pattern, repl [, n])`

Returns a copy of `s` in which all (or the first `n`, if given) occurrences of the `pattern` have been replaced by a replacement string specified by `repl`, which can be a string, a table, or a function. `gsub` also returns, as its second value, the total number of matches that occurred. The name `gsub` comes from *Global SUBstitution*.

If `repl` is a string, then its value is used for replacement. The character `%` works as an escape character: any sequence in `repl` of the form `%`*d*, with *d* between 1 and 9, stands for the value of the *d*-th captured substring. The sequence `%0` stands for the whole match. The sequence `%%` stands for a single `%`.

If `repl` is a table, then the table is queried for every match, using the first capture as the key.

If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order.

In any case, if the pattern specifies no captures, then it behaves as if the whole pattern was inside a capture.

If the value returned by the table query or by the function call is a string or a number, then it is used as the replacement string; otherwise, if it is `false` or `nil`, then there is no replacement (that is, the original match is kept in the string).

Here are some examples:

```
x = string.gsub("hello world","(%w+)","%1 %1")
--> x="hello hello world world"

x = string.gsub("hello world","%w+","%0 %0",1)
--> x="hello hello world"

x = string.gsub("hello world from Lua",
                "(%w+)%s*(%w+)","%2 %1")
--> x="world hello Lua from"

x = string.gsub("home = $HOME, user = $USER",
                "%$(%w+)", os.getenv)
--> x="home = /home/roberto, user = roberto"

x = string.gsub("4+5 = $return 4+5$",
                "%$(.-)%$",
                function (s) return load(s)()
    end)
--> x="4+5 = 9"

local t = {name="lua", version="5.2"}
x = string.gsub("$name-$version.tar.gz",
                "%$(%w+)",t)
--> x="lua-5.2.tar.gz"
```

`string.len (s)`

Receives a string and returns its length. The empty string `""` has length 0. Embedded zeros are counted, so `"a\000bc\000"` has length 5.

`string.lower (s)`

Receives a string and returns a copy of this string with all uppercase letters changed to lowercase. All other characters are left unchanged. The definition of what an uppercase letter is depends on the current locale.

`string.match (s, pattern [, init])`

Looks for the first *match* of `pattern` in the string `s`. If it finds one, then `match` returns the captures from the pattern; otherwise it returns `nil`. If `pattern` specifies no captures, then the whole match is returned. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and can be negative.

`string.rep (s, n [, sep])`

Returns a string that is the concatenation of `n` copies of the string `s` separated by the string `sep`. The default value for `sep` is the empty string (that is, no separator).

`string.reverse (s)`

Returns a string that is the string `s` reversed.

`string.sub (s, i [, j])`

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` can be negative. If `j` is absent, then it is assumed to be -1 (which is the same as the string length). In particular, the call `string.sub(s,1,j)` returns a prefix of `s` with length `j`, and `string.sub(s,-i)` returns a suffix of `s` with length `i`.

If, after the translation of negative indices, `i` is less than 1, it is corrected to 1. If `j` is greater than the string length, it is corrected to that length. If, after these corrections, `i` is greater than `j`, the function returns the empty string.

`string.upper (s)`

Receives a string and returns a copy of this string with all lowercase letters changed to uppercase. All other characters are left unchanged. The definition of what a lowercase letter is depends on the current locale.

6.4.1 · Patterns

Character Class:

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

- *x*: (where *x* is not one of the *magic characters* `^$()%.[]*+-?`) represents the character *x* itself.
- `.`: (a dot) represents all characters.
- `%a`: represents all letters.
- `%c`: represents all control characters.
- `%d`: represents all digits.
- `%g`: represents all printable characters except space.
- `%l`: represents all lowercase letters.
- `%p`: represents all punctuation characters.
- `%s`: represents all space characters.
- `%u`: represents all uppercase letters.
- `%w`: represents all alphanumeric characters.
- `%x`: represents all hexadecimal digits.
- `%`*x*: (where *x* is any non-alphanumeric character) represents the character *x*. This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a `%` when used to represent itself in a pattern.
- `[`*set*`]`: represents the class which is the union of all characters in *set*. A range of characters can be specified by separating the end characters of the range, in ascending order, with a '`-`'. All classes `%`*x* described above can also be used as components in *set*. All other characters in *set* represent themselves. For example, `[%w_]` (or `[_%w]`) represents all alphanumeric characters plus the underscore, `[0-7]` represents the octal digits, and `[0-7%l%-]` represents the octal digits plus the lowercase letters plus the '`-`' character.
  The interaction between ranges and classes is not defined. Therefore, patterns like `[%a-z]` or `[a-%%]` have no meaning.
- `[^`*set*`]`: represents the complement of *set*, where *set* is interpreted as above.

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

The definitions of letter, space, and other character groups depend on the current locale. In particular, the class `[a-z]` may not be equivalent to `%l`.

Pattern Item:

A *pattern item* can be:

- a single character class, which matches any single character in the class;
- a single character class followed by `*`, which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by `+`, which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by `-`, which also matches 0 or more repetitions of characters in the class. Unlike '`*`', these repetition items will always match the shortest possible sequence;
- a single character class followed by `?`, which matches 0 or 1 occurrence of a character in the class;
- `%`*n*, for *n* between 1 and 9; such item matches a substring equal to the *n*-th captured string (see below);
- `%b`*xy*, where *x* and *y* are two distinct characters; such item matches strings that start with *x*, end with *y*, and where the *x* and *y* are *balanced*. This means that, if one reads the string from left to right, counting +1 for an *x* and −1 for a *y*, the ending *y* is the first *y* where the count reaches 0. For instance, the item `%b()` matches expressions with balanced parentheses.
- `%f[`*set*`]`, a *frontier pattern*; such item matches an empty string at any position such that the next character belongs to *set* and the previous character does not belong to *set*. The set *set* is interpreted as previously described. The beginning and the end of the subject are handled as if they were the character '`\0`'.

Pattern:

A *pattern* is a sequence of pattern items. A caret `^` 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.

Captures:

A pattern can contain sub-patterns enclosed in parentheses; they describe *captures*. When a match succeeds, the substrings of the subject string that match captures are stored (*captured*) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern `"(a*(.)%w(%s*))"`, the part of the string matching `"a*(.)%w(%s*)"` is stored as the first capture (and therefore has number 1); the character matching `"."` is captured with number 2, and the part matching `"%s*"` has number 3.

As a special case, the empty capture `()` captures the current string position (a number). For instance, if we apply the pattern `"()aa()"` on the string `"flaaap"`, there will be two captures: 3 and 5.

6.5 · Table Manipulation

This library provides generic functions for table manipulation. It provides all its functions inside the table `table`.

Remember that, whenever an operation needs the length of a table, the table should be a proper sequence or have a `__len` metamethod (see §3.4.6). All functions ignore non-numeric keys in tables given as arguments.

For performance reasons, all table accesses (get/set) performed by these functions are raw.

`table.concat (list [, sep [, i [, j]]])`

Given a list where all elements are strings or numbers,

returns `list[i]..sep..list[i+1]` ··· `sep..list[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is `#list`. If `i` is greater than `j`, returns the empty string.

`table.insert (list, [pos,] value)`

Inserts element `value` at position `pos` in `list`, shifting up the elements `list[pos], list[pos+1], ···, list[#list]`. The default value for `pos` is `#list+1`, so that a call `table.insert(t,x)` inserts `x` at the end of list `t`.

`table.pack (···)`

Returns a new table with all parameters stored into keys 1, 2, etc. and with a field "`n`" with the total number of parameters. Note that the resulting table may not be a sequence.

`table.remove (list [, pos])`

Removes from `list` the element at position `pos`, shifting down the elements `list[pos+1], list[pos+2], ···, list[#list]` and erasing element `list[#list]`. Returns the value of the removed element. The default value for `pos` is `#list`, so that a call `table.remove(t)` removes the last element of list `t`.

`table.sort (list [, comp])`

Sorts list elements in a given order, *in-place*, from `list[1]` to `list[#list]`. If `comp` is given, then it must be a function that receives two list elements and returns true when the first element must come before the second in the final order (so that `not comp(list[i+1],list[i])` will be true after the sort). If `comp` is not given, then the standard Lua operator `<` is used instead.

The sort algorithm is not stable; that is, elements considered equal by the given order may have their relative positions changed by the sort.

`table.unpack (list [, i [, j]])`

Returns the elements from the given table. This function is equivalent to

```
return list[i], list[i+1], ···, list[j]
```

By default, `i` is 1 and `j` is `#list`.

6.6 · Mathematical Functions

This library is an interface to the standard C math library. It provides all its functions inside the table `math`.

`math.abs (x)`

Returns the absolute value of `x`.

`math.acos (x)`

Returns the arc cosine of `x` (in radians).

`math.asin (x)`

Returns the arc sine of `x` (in radians).

`math.atan (x)`

Returns the arc tangent of `x` (in radians).

`math.atan2 (y, x)`

Returns the arc tangent of `y/x` (in radians), but uses the signs of both parameters to find the quadrant of the result. (It also handles correctly the case of `x` being zero.)

`math.ceil (x)`

Returns the smallest integer larger than or equal to `x`.

`math.cos (x)`

Returns the cosine of `x` (assumed to be in radians).

`math.cosh (x)`

Returns the hyperbolic cosine of `x`.

`math.deg (x)`

Returns the angle `x` (given in radians) in degrees.

`math.exp (x)`

Returns the value e^x.

`math.floor (x)`

Returns the largest integer smaller than or equal to `x`.

`math.fmod(x,y)`

Returns the remainder of the division of `x` by `y` that rounds the quotient towards zero.

`math.frexp(x)`

Returns m and e such that $x = m2^e$, e is an integer and the absolute value of m is in the range $[0.5, 1)$ (or zero when `x` is zero).

`math.huge`

The value `HUGE_VAL`, a value larger than or equal to any other numerical value.

`math.ldexp(m,e)`

Returns m^e (`e` should be an integer).

`math.log(x[,base])`

Returns the logarithm of `x` in the given base. The default for `base` is e (so that the function returns the natural logarithm of `x`).

`math.max(x,···)`

Returns the maximum value among its arguments.

`math.min(x,···)`

Returns the minimum value among its arguments.

`math.modf (x)`

Returns two numbers, the integral part of `x` and the fractional part of `x`.

`math.pi`

The value of π.

`math.pow(x,y)`

Returns x^y. (You can also use the expression `x^y` to compute this value.)

`math.rad(x)`

Returns the angle `x` (given in degrees) in radians.

`math.random([m[,n]])`

This function is an interface to the simple pseudo-random generator function `rand` provided by Standard C. (No guarantees can be given for its statistical properties.)

When called without arguments, returns a uniform pseudo-random real number in the range $[0, 1)$. When called with an integer number `m`, `math.random` returns a uniform pseudo-random integer in the range $[1, m]$. When called with two integer numbers `m` and `n`, `math.random` returns a uniform pseudo-random integer in the range $[m, n]$.

`math.randomseed(x)`

Sets `x` as the "seed" for the pseudo-random generator: equal seeds produce equal sequences of numbers.

`math.sin(x)`

Returns the sine of `x` (assumed to be in radians).

`math.sinh(x)`

Returns the hyperbolic sine of `x`.

`math.sqrt(x)`

Returns the square root of `x`. (You can also use the expression `x^0.5` to compute this value.)

`math.tan (x)`

Returns the tangent of `x` (assumed to be in radians).

`math.tanh (x)`

Returns the hyperbolic tangent of `x`.

6.7 · Bitwise Operations

This library provides bitwise operations. It provides all its functions inside the table `bit32`.

Unless otherwise stated, all functions accept numeric arguments in the range $(-2^{51}, +2^{51})$; each argument is normalized to the remainder of its division by 2^{32} and truncated to an integer (in some unspecified way), so that its final value falls in the range $[0, 2^{32} - 1]$. Similarly, all results are in the range $[0, 2^{32} - 1]$. Note that `bit32.bnot(0)` is `0xFFFFFFFF`, which is different from `-1`.

`bit32.arshift (x, disp)`

Returns the number `x` shifted `disp` bits to the right. The number `disp` may be any representable integer. Negative displacements shift to the left.

This shift operation is what is called arithmetic shift. Vacant bits on the left are filled with copies of the higher bit of `x`; vacant bits on the right are filled with zeros. In particular, displacements with absolute values higher than 31 result in zero or `0xFFFFFFFF` (all original bits are shifted out).

`bit32.band(···)`

Returns the bitwise *and* of its operands.

`bit32.bnot (x)`

Returns the bitwise negation of `x`. For any integer `x`, the following identity holds:

```
assert(bit32.bnot(x) == (-1 - x) % 2^32)
```

`bit32.bor(···)`

Returns the bitwise *or* of its operands.

`bit32.btest(···)`

Returns a boolean signaling whether the bitwise *and* of its operands is different from zero.

`bit32.bxor(···)`

Returns the bitwise *exclusive or* of its operands.

`bit32.extract(n,field[,width])`

Returns the unsigned number formed by the bits `field` to `field+width-1` from `n`. Bits are numbered from 0 (least significant) to 31 (most significant). All accessed bits must be in the range $[0, 31]$.

The default for `width` is 1.

`bit32.replace (n, v, field [, width])`

Returns a copy of `n` with the bits `field` to `field + width - 1` replaced by the value `v`. See `bit32.extract` for details about `field` and `width`.

`bit32.lrotate (x, disp)`

Returns the number `x` rotated `disp` bits to the left. The number `disp` may be any representable integer.

For any valid displacement, the following identity holds:

```
assert(bit32.lrotate(x, disp) ==
       bit32.lrotate(x, disp % 32))
```

In particular, negative displacements rotate to the right.

`bit32.lshift (x, disp)`

Returns the number `x` shifted `disp` bits to the left. The number `disp` may be any representable integer. Negative displacements shift to the right. In any direction, vacant bits are filled with zeros. In particular, displacements with absolute values higher than 31 result in zero (all bits are shifted out).

For positive displacements, the following equality holds:

```
assert(bit32.lshift(b, disp) ==
       (b * 2^disp) % 2^32)
```

`bit32.rrotate (x, disp)`

Returns the number `x` rotated `disp` bits to the right. The number `disp` may be any representable integer.

For any valid displacement, the following identity holds:

```
assert(bit32.rrotate(x, disp) ==
       bit32.rrotate(x, disp % 32))
```

In particular, negative displacements rotate to the left.

`bit32.rshift (x, disp)`

Returns the number `x` shifted `disp` bits to the right. The number `disp` may be any representable integer. Negative displacements shift to the left. In any direction, vacant bits are filled with zeros. In particular, displacements with absolute values higher than 31 result in zero (all bits are shifted out).

For positive displacements, the following equality holds:

```
assert(bit32.rshift(b, disp) ==
       math.floor(b % 2^32 / 2^disp))
```

This shift operation is what is called logical shift.

6.8 · Input and Output Facilities

The I/O library provides two different styles for file manipulation. The first one uses implicit file descriptors; that is, there are operations to set a default input file and a default output file, and all input/output operations are over these default files. The second style uses explicit file descriptors.

When using implicit file descriptors, all operations are supplied by table `io`. When using explicit file descriptors, the operation `io.open` returns a file descriptor and then all operations are supplied as methods of the file descriptor.

The table `io` also provides three predefined file descriptors with their usual meanings from C: `io.stdin`, `io.stdout`, and `io.stderr`. The I/O library never closes these files.

Unless otherwise stated, all I/O functions return `nil` on failure (plus an error message as a second result and a system-dependent error code as a third result) and some value different from `nil` on success.

`io.close ([file])`

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

`io.flush ()`

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

`io.input ([file])`

When called with a file name, it opens the named file (in text mode), and sets its handle as the default input file. When called with a file handle, it simply sets this file handle as the default input file. When called without parameters, it returns the current default input file.

In case of errors this function raises the error, instead of returning an error code.

`io.lines ([filename ···])`

Opens the given file name in read mode and returns an iterator function that works like `file:lines(···)` over the opened file. When the iterator function detects the end of file, it returns `nil` (to finish the loop) and automatically closes the file.

The call `io.lines()` (with no file name) is equivalent to `io.input():lines()`; that is, it iterates over the lines of the default input file. In this case it does not close the file when the loop ends.

In case of errors this function raises the error, instead of returning an error code.

`io.open (filename [, mode])`

This function opens a file, in the mode specified in the string `mode`. It returns a new file handle, or, in case of errors, `nil` plus an error message.

The `mode` string can be any of the following:

- `"r"`: read mode (the default);
- `"w"`: write mode;

- "a": append mode;
- "r+": update mode, all previous data is preserved;
- "w+": update mode, all previous data is erased;
- "a+": append update mode, previous data is preserved, writing is only allowed at the end of file.

The mode string can also have a b at the end, which is needed in some systems to open the file in binary mode.

`io.output ([file])`

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

`io.popen (prog [, mode])`

This function is system dependent and is not available on all platforms.

Starts program prog in a separated process and returns a file handle that you can use to read data from this program (if mode is "r", the default) or to write data to this program (if mode is "w").

`io.read (···)`

Equivalent to io.input():read(···).

`io.tmpfile ()`

Returns a handle for a temporary file. This file is opened in update mode and it is automatically removed when the program ends.

`io.type (obj)`

Checks whether obj is a valid file handle. Returns the string "file" if obj is an open file handle, "closed file" if obj is a closed file handle, or nil if obj is not a file handle.

`io.write (···)`

Equivalent to io.output():write(···).

`file:close ()`

Closes file. Note that files are automatically closed when their handles are garbage collected, but that takes an unpredictable amount of time to happen.

When closing a file handle created with io.popen, file:close returns the same values returned by os.execute.

`file:flush ()`

Saves any written data to file.

`file:lines (···)`

Returns an iterator function that, each time it is called, reads the file according to the given formats. When no format is given, uses "*l" as a default. As an example, the construction

```
for c in file:lines(1) do body end
```

will iterate over all characters of the file, starting at the current position. Unlike io.lines, this function does not close the file when the loop ends.

In case of errors this function raises the error, instead of returning an error code.

`file:read (···)`

Reads the file file, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or nil if it cannot read data with the specified format. When called without formats, it uses a default format that reads the next line (see below).

The available formats are

- "*n": reads a number; this is the only format that returns a number instead of a string.
- "*a": reads the whole file, starting at the current position. On end of file, it returns the empty string.
- "*l": reads the next line skipping the end of line, returning nil on end of file. This is the default format.
- "*L": reads the next line keeping the end of line (if present), returning nil on end of file.
- *number*: reads a string with up to this number of bytes, returning nil on end of file. If number is zero, it reads nothing and returns an empty string, or nil on end of file.

`file:seek ([whence [, offset]])`

Sets and gets the file position, measured from the beginning of the file, to the position given by offset plus 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;

In case of success, seek returns the final file position, measured in bytes from the beginning of the file. If seek fails, it returns nil, plus a string describing the error.

The default value for whence is "cur", and for offset is 0. Therefore, the call file:seek() returns the current file position, without changing it; the call file:seek("set") sets the position to the beginning of the file (and returns 0); and the call file:seek("end") sets the position to the end of the file, and returns its size.

`file:setvbuf (mode [, size])`

Sets the buffering mode for an output file. There are three available modes:

- "no": no buffering; the result of any output operation appears immediately.
- "full": full buffering; output operation is performed only when the buffer is full or when you explicitly flush the file (see io.flush).
- "line": line buffering; output is buffered until a newline is output or there is any input from some special files (such as a terminal device).

For the last two cases, `size` specifies the size of the buffer, in bytes. The default is an appropriate size.

`file:write (···)`

Writes the value of each of its arguments to `file`. The arguments must be strings or numbers.

In case of success, this function returns `file`. Otherwise it returns `nil` plus a string describing the error.

6.9 · Operating System Facilities

This library is implemented through table `os`.

`os.clock ()`

Returns an approximation of the amount in seconds of CPU time used by the program.

`os.date ([format [, time]])`

Returns a string or a table containing date and time, formatted according to the given string `format`.

If the `time` argument is present, this is the time to be formatted (see the `os.time` function for a description of this value). Otherwise, `date` formats the current time.

If `format` starts with `!`, then the date is formatted in Coordinated Universal Time. After this optional character, if `format` is the string `"*t"`, then `date` returns a table with the following fields: `year` (four digits), `month` (1—12), `day` (1—31), `hour` (0—23), `min` (0—59), `sec` (0—61), `wday` (weekday, Sunday is 1), `yday` (day of the year), and `isdst` (daylight saving flag, a boolean). This last field may be absent if the information is not available.

If `format` is not `"*t"`, then `date` returns the date as a string, formatted according to the same rules as the C function `strftime`.

When called without arguments, `date` returns a reasonable date and time representation that depends on the host system and on the current locale (that is, `os.date()` is equivalent to `os.date("%c")`).

On some systems, this function may be not thread safe.

`os.difftime (t2, t1)`

Returns the number of seconds from time `t1` to time `t2`. In POSIX, Windows, and some other systems, this value is exactly `t2-t1`.

`os.execute ([command])`

This function is equivalent to the C function `system`. It passes `command` to be executed by an operating system shell. Its first result is `true` if the command terminated successfully, or `nil` otherwise. After this first result the function returns a string and a number, as follows:

- `"exit"`: the command terminated normally; the following number is the exit status of the command.
- `"signal"`: the command was terminated by a signal; the following number is the signal that terminated the command.

When called without a `command`, `os.execute` returns a boolean that is true if a shell is available.

`os.exit ([code [, close]])`

Calls the C function `exit` to terminate the host program. If `code` is `true`, the returned status is `EXIT_SUCCESS`; if `code` is `false`, the returned status is `EXIT_FAILURE`; if `code` is a number, the returned status is this number. The default value for `code` is `true`.

If the optional second argument `close` is true, closes the Lua state before exiting.

`os.getenv (varname)`

Returns the value of the process environment variable `varname`, or `nil` if the variable is not defined.

`os.remove (filename)`

Deletes the file (or empty directory, on POSIX systems) with the given name. If this function fails, it returns `nil`, plus a string describing the error and the error code.

`os.rename (oldname, newname)`

Renames file or directory named `oldname` to `newname`. If this function fails, it returns `nil`, plus a string describing the error and the error code.

`os.setlocale (locale [, category])`

Sets the current locale of the program. `locale` is a system-dependent string specifying a locale; `category` is an optional string describing which category to change: `"all"`, `"collate"`, `"ctype"`, `"monetary"`, `"numeric"`, or `"time"`; the default category is `"all"`. The function returns the name of the new locale, or `nil` if the request cannot be honored.

If `locale` is the empty string, the current locale is set to an implementation-defined native locale. If `locale` is the string `"C"`, the current locale is set to the standard C locale.

When called with `nil` as the first argument, this function only returns the name of the current locale for the given category.

`os.time ([table])`

Returns the current time when called without arguments, or a time representing the date and time specified by the given table. This table must have fields `year`, `month`, and `day`, and may have fields `hour` (default is 12), `min` (default is 0), `sec` (default is 0), and `isdst` (default is `nil`). For a description of these fields, see the `os.date` function.

The returned value is a number, whose meaning depends on your system. In POSIX, Windows, and some other systems, this number counts the number of seconds since some given start time (the "epoch"). In other systems, the meaning is not specified, and the number returned by `time` can be used only as an argument to `os.date` and `os.difftime`.

`os.tmpname ()`

Returns a string with a file name that can be used for a temporary file. The file must be explicitly opened before its use and explicitly removed when no longer needed.

On POSIX systems, this function also creates a file with that name, to avoid security risks. (Someone else might create the file with wrong permissions in the time between getting the name and creating the file.) You still have to open the file to use it and to remove it (even if you do not use it).

When possible, you may prefer to use `io.tmpfile`, which automatically removes the file when the program ends.

6.10 · The Debug Library

This library provides the functionality of the debug interface (§4.9) to Lua programs. You should exert care when using this library. Several of its functions violate basic assumptions about Lua code (e.g., that variables local to a function cannot be accessed from outside; that userdata metatables cannot be changed by Lua code; that Lua programs do not crash) and therefore can compromise otherwise secure code. Moreover, some functions in this library may be slow.

All functions in this library are provided inside the `debug` table. All functions that operate over a thread have an optional first argument which is the thread to operate over. The default is always the current thread.

`debug.debug ()`

Enters an interactive mode with the user, running each string that the user enters. Using simple commands and other debug facilities, the user can inspect global and local variables, change their values, evaluate expressions, and so on. A line containing only the word `cont` finishes this function, so that the caller continues its execution.

Note that commands for `debug.debug` are not lexically nested within any function and so have no direct access to local variables.

`debug.gethook ([thread])`

Returns the current hook settings of the thread, as three values: the current hook function, the current hook mask, and the current hook count (as set by the `debug.sethook` function).

`debug.getinfo ([thread,] f [, what])`

Returns a table with information about a function. You can give the function directly or you can give a number as the value of `f`, which means the function running at level `f` of the call stack of the given thread: level 0 is the current function (`getinfo` itself); level 1 is the function that called `getinfo` (except for tail calls, which do not count on the stack); and so on. If `f` is a number larger than the number of active functions, then `getinfo` returns `nil`.

The returned table can contain all the fields returned by `lua_getinfo`, with the string `what` describing which fields to fill in. The default for `what` is to get all information available, except the table of valid lines. If present, the option '`f`' adds a field named `func` with the function itself. If present, the option '`L`' adds a field named `activelines` with the table of valid lines.

For instance, the expression `debug.getinfo(1,"n").name` returns a table with a name for the current function, if a reasonable name can be found, and the expression `debug.getinfo(print)` returns a table with all available information about the `print` function.

`debug.getlocal ([thread,] f, local)`

This function returns the name and the value of the local variable with index `local` of the function at level `f` of the stack. This function accesses not only explicit local variables, but also parameters, temporaries, etc.

The first parameter or local variable has index 1, and so on, until the last active variable. Negative indices refer to vararg parameters; -1 is the first vararg parameter. The function returns `nil` if there is no variable with the given index, and raises an error when called with a level out of range. (You can call `debug.getinfo` to check whether the level is valid.)

Variable names starting with '`(`' (open parentheses) represent internal variables (loop control variables, temporaries, varargs, and C function locals).

The parameter `f` may also be a function. In that case, `getlocal` returns only the name of function parameters.

`debug.getmetatable (value)`

Returns the metatable of the given `value` or `nil` if it does not have a metatable.

`debug.getregistry ()`

Returns the registry table (see §4.5).

`debug.getupvalue (f, up)`

This function returns the name and the value of the upvalue with index `up` of the function `f`. The function returns `nil` if there is no upvalue with the given index.

`debug.getuservalue (u)`

Returns the Lua value associated to `u`. If `u` is not a userdata, returns `nil`.

`debug.sethook ([thread,] hook, mask [, count])`

Sets the given function as a hook. The string `mask` and the number `count` describe when the hook will be called. The string mask may have the following characters, with the given meaning:

- `c`: the hook is called every time Lua calls a function;
- `r`: the hook is called every time Lua returns from a function;

- `l`: the hook is called every time Lua enters a new line of code.

With a `count` different from zero, the hook is called after every `count` instructions.

When called without arguments, `debug.sethook` turns off the hook.

When the hook is called, its first parameter is a string describing the event that has triggered its call: `"call"` (or `"tail call"`), `"return"`, `"line"`, and `"count"`. For line events, the hook also gets the new line number as its second parameter. Inside a hook, you can call `getinfo` with level 2 to get more information about the running function (level 0 is the `getinfo` function, and level 1 is the hook function).

`debug.setlocal ([thread,] level, local, value)`

This function assigns the value `value` to the local variable with index `local` of the function at level `level` of the stack. The function returns `nil` if there is no local variable with the given index, and raises an error when called with a `level` out of range. (You can call `getinfo` to check whether the level is valid.) Otherwise, it returns the name of the local variable.

See `debug.getlocal` for more information about variable indices and names.

`debug.setmetatable (value, table)`

Sets the metatable for the given `value` to the given `table` (which can be `nil`). Returns `value`.

`debug.setupvalue (f, up, value)`

This function assigns the value `value` to the upvalue with index `up` of the function `f`. The function returns `nil` if there is no upvalue with the given index. Otherwise, it returns the name of the upvalue.

`debug.setuservalue (udata, value)`

Sets the given `value` as the Lua value associated to the given `udata`. `value` must be a table or `nil`; `udata` must be a full userdata.

Returns `udata`.

`debug.traceback ([thread,] [message [,level]])`

If `message` is present but is neither a string nor `nil`, this function returns `message` without further processing. Otherwise, it returns a string with a traceback of the call stack. An optional `message` string is appended at the beginning of the traceback. An optional `level` number tells at which level to start the traceback (default is 1, the function calling `traceback`).

`debug.upvalueid (f, n)`

Returns an unique identifier (as a light userdata) for the upvalue numbered `n` from the given function.

These unique identifiers allow a program to check whether different closures share upvalues. Lua closures that share an upvalue (that is, that access a same external local variable) will return identical ids for those upvalue indices.

`debug.upvaluejoin (f1, n1, f2, n2)`

Make the `n1`-th upvalue of the Lua closure `f1` refer to the `n2`-th upvalue of the Lua closure `f2`.

7 · Lua Standalone

Although Lua has been designed as an extension language, to be embedded in a host C program, it is also frequently used as a standalone language. An interpreter for Lua as a standalone language, called simply `lua`, is provided with the standard distribution. The standalone interpreter includes all standard libraries, including the debug library. Its usage is:

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

The options are:

- `-e` *stat*: executes string *stat*;
- `-l` *mod*: '`requires`' *mod*;
- `-i`: enters interactive mode after running a script;
- `-v`: prints version information;
- `-E`: ignores environment variables;
- `--`: stops handling options;
- `-`: executes `stdin` as a file and stops handling options.

After handling its options, `lua` runs the given script, passing to it the given `args` as string arguments. When called without arguments, `lua` behaves as `lua -v -i` when the standard input (`stdin`) is a terminal, and as `lua -` otherwise.

When called without option `-E`, the interpreter checks for an environment variable `LUA_INIT_5_2` (or `LUA_INIT` if it is not defined) before running any argument. If the variable content has the format `@filename`, then `lua` executes the file. Otherwise, `lua` executes the string itself.

When called with option `-E`, besides ignoring `LUA_INIT`, Lua also ignores the values of `LUA_PATH` and `LUA_CPATH`, setting the values of `package.path` and `package.cpath` with the default paths defined in `luaconf.h`.

All options are handled in order, except `-i` and `-E`. For instance, an invocation like

```
$ lua -e'a=1' -e 'print(a)' script.lua
```

will first set `a` to 1, then print the value of `a`, and finally run the file `script.lua` with no arguments. (Here `$` is the shell prompt. Your prompt may be different.)

Before starting to run the script, `lua` collects all arguments in the command line in a global table called `arg`. The script name is stored at index 0, the first argument after the script name goes to index 1, and so on. Any arguments before the script name (that is, the interpreter name plus the options) go to negative indices. For instance, in the call

```
$ lua -la b.lua t1 t2
```

the interpreter first runs the file `a.lua`, then creates a table

```
arg = { [-2] = "lua", [-1] = "-la",
        [0] = "b.lua",
        [1] = "t1", [2] = "t2" }
```

and finally runs the file `b.lua`. The script is called with `arg[1]`, `arg[2]`, ... as arguments; it can also access these arguments with the vararg expression `...`.

In interactive mode, if you write an incomplete statement, the interpreter waits for its completion by issuing a different prompt.

In case of unprotected errors in the script, the interpreter reports the error to the standard error stream. If the error object is a string, the interpreter adds a stack traceback to it. Otherwise, if the error object has a metamethod `__tostring`, the interpreter calls this metamethod to produce the final message. Finally, if the error object is `nil`, the interpreter does not report the error.

When finishing normally, the interpreter closes its main Lua state (see `lua_close`). The script can avoid this step by calling `os.exit` to terminate.

To allow the use of Lua as a script interpreter in Unix systems, the standalone interpreter skips the first line of a chunk if it starts with `#`. Therefore, Lua scripts can be made into executable programs by using `chmod +x` and the `#!` form, as in

```
#!/usr/local/bin/lua
```

(Of course, the location of the Lua interpreter may be different in your machine. If `lua` is in your `PATH`, then

```
#!/usr/bin/env lua
```

is a more portable solution.)

8 · Incompatibilities with the Previous Version

Here we list the incompatibilities that you may find when moving a program from Lua 5.1 to Lua 5.2. You can avoid some incompatibilities by compiling Lua with appropriate options (see file `luaconf.h`). However, all these compatibility options will be removed in the next version of Lua. Similarly, all features marked as deprecated in Lua 5.1 have been removed in Lua 5.2.

8.1 · Changes in the Language

- The concept of *environment* changed. Only Lua functions have environments. To set the environment of a Lua function, use the variable `_ENV` or the function `load`.
  C functions no longer have environments. Use an upvalue with a shared table if you need to keep shared state among several C functions. (You may use `luaL_setfuncs` to open a C library with all functions sharing a common upvalue.)
  To manipulate the "environment" of a userdata (which is now called user value), use the new functions `lua_getuservalue` and `lua_setuservalue`.
- Lua identifiers cannot use locale-dependent letters.
- Doing a step or a full collection in the garbage collector does not restart the collector if it has been stopped.
- Weak tables with weak keys now perform like *ephemeron tables*.
- The event *tail return* in debug hooks was removed. Instead, tail calls generate a special new event, *tail call*, so that the debugger can know that there will not be a corresponding return event.
- Equality between function values has changed. Now, a function definition may not create a new value; it may reuse some previous value if there is no observable difference to the new function.

8.2 · Changes in the Libraries

- Function `module` is deprecated. It is easy to set up a module with regular Lua code. Modules are not expected to set global variables.
- Functions `setfenv` and `getfenv` were removed, because of the changes in environments.
- Function `math.log10` is deprecated. Use `math.log` with 10 as its second argument, instead.
- Function `loadstring` is deprecated. Use `load` instead; it now accepts string arguments and are exactly equivalent to `loadstring`.
- Function `table.maxn` is deprecated. Write it in Lua if you really need it.
- Function `os.execute` now returns `true` when command terminates successfully and `nil` plus error information otherwise.
- Function `unpack` was moved into the table library and therefore must be called as `table.unpack`.
- Character class `%z` in patterns is deprecated, as now patterns may contain '\0' as a regular character.
- The table `package.loaders` was renamed `package.searchers`.
- Lua does not have bytecode verification anymore. So, all functions that load code (`load` and `loadfile`) are potentially insecure when loading untrusted binary data. (Actually, those functions were already insecure because of flaws in the verification algorithm.) When in doubt, use the `mode` argument of those functions to restrict them to loading textual chunks.
- The standard paths in the official distribution may change between versions.

8.3 · Changes in the API

- Pseudoindex `LUA_GLOBALSINDEX` was removed. You must get the global environment from the registry (see §4.5).
- Pseudoindex `LUA_ENVIRONINDEX` and functions `lua_getfenv`/`lua_setfenv` were removed, as C functions no longer have environments.
- Function `luaL_register` is deprecated. Use `luaL_setfuncs` so that your module does not create globals. (Modules are not expected to set global variables anymore.)

- The `osize` argument to the allocation function may not be zero when creating a new block, that is, when `ptr` is `NULL` (see `lua_Alloc`). Use only the test `ptr == NULL` to check whether the block is new.
- Finalizers (`__gc` metamethods) for userdata are called in the reverse order that they were marked for finalization, not that they were created (see §2.5.1). (Most userdata are marked immediately after they are created.) Moreover, if the metatable does not have a `__gc` field when set, the finalizer will not be called, even if it is set later.
- `luaL_typerror` was removed. Write your own version if you need it.
- Function `lua_cpcall` is deprecated. You can simply push the function with `lua_pushcfunction` and call it with `lua_pcall`.
- Functions `lua_equal` and `lua_lessthan` are deprecated. Use the new `lua_compare` with appropriate options instead.
- Function `lua_objlen` was renamed `lua_rawlen`.
- Function `lua_load` has an extra parameter, `mode`. Pass `NULL` to simulate the old behavior.
- Function `lua_resume` has an extra parameter, `from`. Pass `NULL` or the thread doing the call.

9 · The Complete Syntax of Lua

Here is the complete syntax of Lua in extended BNF. (It does not describe operator precedences.)

```
chunk ::= block
block ::= {stat} [retstat]
stat ::=  ‘;’ |
        varlist ‘=’ explist |
        functioncall |
        label |
        break |
        goto Name |
        do block end |
        while exp do block end |
        repeat block until exp |
        if exp then block
          {elseif exp then block}
          [else block] end |
        for Name ‘=’ exp ‘,’ exp [‘,’ exp]
          do block end |
        for namelist in explist
          do block end |
        function funcname funcbody |
        local function Name funcbody |
        local namelist [‘=’ explist]
retstat ::= return [explist] [‘;’]
label ::= ‘::’ Name ‘::’
funcname ::= Name {‘.’ Name} [‘:’ Name]
varlist ::= var {‘,’ var}
var ::=  Name | prefixexp ‘[’ exp ‘]’ |
        prefixexp ‘.’ Name
namelist ::= Name {‘,’ Name}
explist ::= exp {‘,’ exp}
exp ::=  nil | false | true | Number |
        String | ‘...’ | functiondef |
        prefixexp | tableconstructor |
        exp binop exp | unop exp
prefixexp ::= var | functioncall | ‘(’ exp ‘)’
functioncall ::=  prefixexp args |
                  prefixexp ‘:’ Name args
args ::=  ‘(’ [explist] ‘)’ |
          tableconstructor | String
functiondef ::= function funcbody
funcbody ::= ‘(’ [parlist] ‘)’ block end
parlist ::= namelist [‘,’ ‘...’] | ‘...’
tableconstructor ::= ‘{’ [fieldlist] ‘}’
fieldlist ::= field {fieldsep field}
                    [fieldsep]
field ::= ‘[’ exp ‘]’ ‘=’ exp |
          Name ‘=’ exp | exp
fieldsep ::= ‘,’ | ‘;’
binop ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘%’ |
          ‘..’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ |
          ‘==’ | ‘~=’ | and | or
unop ::= ‘-’ | not | ‘#’
```