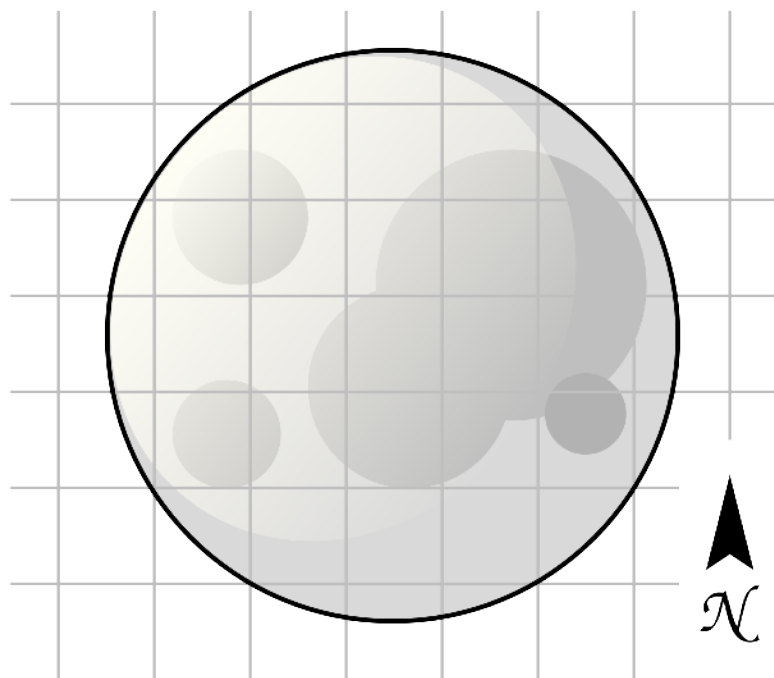


Lua

Quick Reference

Covers
Lua
5.4, 5.3,
5.2, and
5.1!

A small, fast, powerful, and embeddable language



Second Edition

Mitchell

Lua Quick Reference

Lua is a small, fast, powerful, and embeddable scripting language. It is well-suited for use in video games, application scripting, embedded devices, and nearly anywhere else a scripting language is needed. This quick reference contains a wealth of knowledge on how to program in and embed Lua, whether it is Lua 5.4, 5.3, 5.2, or 5.1. It groups the language's features and C API in a convenient and easy-to-use manner, while clearly marking the differences between Lua versions.

This book covers:

- Lua syntax, expressions, and statements
- Metatables and metamethods
- Object-oriented programming with Lua
- Creating and working with Lua and C Modules
- Lua's standard library and its C API
- Collaborative multi-threading in Lua and C
- How to embed and use Lua within a host
- And much more

Mitchell commands more than 15 years of experience programming and embedding Lua in both the corporate and open-source realms.

Triple Quasar Books

SECOND EDITION

Lua

Quick Reference

Mitchell

Lua Quick Reference

by Mitchell

Copyright © 2017, 2020 Mitchell. All rights reserved.

Contact the author at books@triplequasar.com.

Although great care has been taken in preparing this book, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. All product names mentioned in this book are trademarks of their respective owners.

Editor: Ana Balan

Technical Reviewer: Robert Gieseke

Cover Designer: Mitchell

Interior Designer: Mitchell

Indexer: Mitchell

Printing history:

July 2017: First Edition

May 2020: Second Edition

ISBN: 978-0-9912379-5-1

Preface to the Second Edition

This book is an updated version of *Lua Quick Reference* and covers many of the features, changes, and incompatibilities introduced in Lua 5.4. Among the new content contained in this edition:

- Local variable attributes, including const and to-be-closed variables.
- Defining to-be-closed variable behavior.
- Lua's new warning system.
- The improved random number generator.
- Changes to Lua's string and thread facilities.
- Compiling Lua programs.
- More details about user values in the Lua C API.
- Various Lua C API additions and changes, including incompatible changes to the threading API.

The book's code examples have also been updated to use Lua 5.4 where applicable.

Contents

Introduction	1
Download	2
Code Editors	2
Conventions	3
Terminology	3
Environment Variables	4
Command Line Options	5
Part I: The Lua Language	
Fundamentals	9
Comments	9
Identifiers and Reserved Words	9
Variables and Scopes	10
Types	12
Nil	12
Booleans	12
Numbers	12
Strings	13
Functions	14
Tables	15
Threads	15
Userdata	16
Perform Basic Value Operations	16
Expressions and Operators	17
Arithmetic Operators	17
Relational Operators	18
Logical Operators	19

Bitwise Operators	20
Other Operators	21
Statements	23
Variable Assignment	23
Control Structures	24
Labels and Goto	27
Functions	27
Functions with Variable Arguments	30
Metatables and Metamethods	32
Assign and Retrieve Metatables	32
Arithmetic Metamethods	33
Relational Metamethods	33
Bitwise Metamethods	34
Closing Metamethod	35
Other Operator and Statement Metamethods	35
Function Metamethods	37
Bypass Metamethods	38
Object-Oriented Programming	38
Define a Class	38
Utilize a Class	41
Modules	42
Create a Lua Module	45
Environments	46
Error Handling and Warnings	48
Load and Run Dynamic Code	49
Numeric Facilities	51
Trigonometric Functions	52

Exponential and Logarithmic Functions	52
Generate Random Numbers	53
Work with Integers	53
String Facilities	54
Create a Compound String	54
Query and Transform Strings	59
Search and Replace Within a String	60
Work with UTF-8 Strings	63
Table and List Facilities	64
Iterate Over a Table	64
Manipulate Lists	66
Unpack Lists	66
Create Strings from Lists	67
Thread Facilities	67
Create a Thread	70
Start, Resume, and Yield a Thread	70
Query Thread Status	71
Close a Thread	71
Input and Output Facilities	72
Simple Input and Output	73
Object-Oriented Input and Output	75
Manage Files	77
Start and Interact with a Process	78
Operating System Facilities	78
Dates and Times	79
Locale Settings	81
Memory Management	82
Miscellaneous	82

Part II: The Lua C API

C API Introduction	87
Compiling Lua Programs	89
The Stack	90
Increase Stack Size	91
Work with Stack Indices	91
Push Values	92
Pop Values	101
Query Values	101
Retrieve Values	102
Basic Stack Operations	105
Element Operations	105
Global Variable Operations	106
Arithmetic Operations	106
Relational Operations	107
Bitwise Operations	107
String Operations	108
Table Operations	109
Length Operations	112
Reference Operations	113
C Functions	114
Define a C Function	114
Register a C Function	120
Call a C Function	121
Metatables	122
Create or Fetch a Metatable	123
Assign a Metatable	123
Retrieve a Metatable	124
Metamethods and Metafields	124

C Modules	124
Error and Warning Handling	129
Retrieve Error Information	130
Load and Run Dynamic Code	130
Threading in C	133
Create a Thread	136
Start or Resume a Thread	136
Yield a Thread	137
Transfer Values Between Threads	139
Query a Thread's Status	139
Close a Thread	139
Call a Function that Yields	139
Memory Management	142
Miscellaneous	143
Lua API Index	145
Concept Index	151

Introduction

Lua¹ is a small, fast, powerful, and embeddable scripting language. It is well-suited for use in video games, application scripting, embedded devices, and nearly anywhere else a scripting language is needed.

Weighing in at just 400KB in compiled size and comprising less than 15,000 lines of highly portable ISO (ANSI) C source code, Lua is a very small language that compiles unmodified on nearly any platform with a C compiler. Many independent benchmarks recognize Lua as one of the fastest scripting languages available. Not only is Lua small and fast, but it is also powerful. With dynamic typing, lexical scoping, first-class functions, collaborative multi-threading, automatic memory management, and an incredibly flexible data structure, Lua is a truly effective object-oriented language, functional language, and data-driven language.

In addition to its assets of size, speed, and power, Lua's primary strength is that it is an embedded language. Lua is implemented as a C library, so a host program can use Lua's C Application Programming Interface (C API) to initialize and interact with a Lua interpreter, define global variables, register C functions that Lua can call, call user-defined Lua functions, and execute arbitrary Lua code. (In fact, Lua's stand-alone interpreter is just a C application that makes use of the Lua library and its C API.) This tight coupling between Lua and its host allows each language to leverage its own strengths: C's raw speed and ability to interact with third-party software, and Lua's flexibility, rapid prototyping, and ease of use.

Lua Quick Reference is designed to help the software developer “get things done” when it comes to programming in and embedding Lua, whether it is Lua 5.4, 5.3, 5.2, or 5.1. This book can even be used with LuaJIT,² a Just-In-Time compiler for Lua based on Lua 5.1. *Lua Quick Reference*'s pragmatic approach assumes the developer has a basic understanding of programming concepts. While familiarity with Lua is helpful, it is not a requirement—this book is suitable for helping seasoned developers quickly get up to speed with the language.

1 <http://www.lua.org>

2 <http://luajit.org/luajit.html>

This quick reference is broken up into two parts: Part I covers the Lua language itself and Part II covers Lua's C API. Each part has a number of descriptive sections with conveniently grouped tasks that cover nearly every aspect of Lua and its C API, with differences between versions clearly marked. For the most part, the contents of each task are not listed in conceptual order. They are listed in procedural order, an order the developer would likely follow when programming in or embedding Lua.

While this book aims to be a complete reference, it does omit some of the lesser-known parts of Lua. For example, this reference does not cover Lua's debug interface, weak tables, or some of the finer details of how external modules are loaded. *Lua Quick Reference* serves as a complement to each Lua version's Reference Manual.

Finally, all code examples in this book are based on Lua 5.4, so adapting them for Lua 5.1, 5.2, and Lua 5.3 may be necessary.

Download

Lua is free software and is available in source format from its website: <http://www.lua.org/download.html>. Links to platform-specific binaries are also available from that page. Lua is highly extensible and can be configured by modifying its *lua conf.h* file prior to compiling the library. For example, on more restricted platforms and embedded devices, the flag "LUA_32BITS" can be defined in order to force Lua to use 32-bit integers and 32-bit floating point numbers.

Code Editors

Programming in Lua does not require an Integrated Development Environment (IDE). A simple text editor is sufficient. The author recommends Textadept,³ a fast, minimalist, and remarkably extensible cross-platform text editor that has fantastic support for Lua. Not only is Textadept free and open-source, but it is also one of the few cross-platform editors that have both a graphical and terminal user interface, the latter being helpful for working on remote machines.

3 <https://orbitalquark.github.io/textadept>

Fundamentals

Lua is a free-form language with whitespace being significant only between identifiers and keywords. Lua source code files typically have the extension *“.lua”*.

Comments

Lua has both line comments and block comments. Line comments start with “--” and apply until the end of the line they occur on. Block comments start with “--[” and end with “]”. Block comment delimiters can contain an optional, equal number of ‘=’ characters between the brackets:

```
-- Line comment.
i = 1 -- another line comment

--[Multi-line
block comment.]
t = {1, 2, --[[in-line block comment]] 3}

--[=[ Block comment that contains ""]]. ]=]
```

Identifiers and Reserved Words

Identifiers are names of variables, table fields, and labels[†]. They are case-sensitive, and can be any combination of ASCII letters, digits, and underscores, though they cannot start with a digit or be a reserved word. Table 1 lists Lua’s reserved words.

Some examples of valid identifiers are “a”, “_”, “A_i”, “a1”, and “END”. Some examples of invalid identifiers are “1a”, “μ”, “function”, and “\$amount”.

NOTE

By convention, identifiers comprising an underscore followed by one or more upper-case letters (e.g. “_M” and “_VERSION”) are reserved for use by Lua itself.

[†] Except for Lua 5.1, which does not have labels.

Table 1. Reserved words

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

^a Not in Lua 5.1.

Variables and Scopes

Lua has both global and local variables. Global variables do not need to be declared; they can simply be used. Local variables must be declared with the keyword “**local**” (unless they are function arguments or **for** loop iterator variables, in which case they are implicitly local). Local variable declarations do not have to specify an initial value.

CAUTION

Variables in Lua are global by default. Lua will never raise an error if an attempt is made to reference a global variable with no previously defined value. Instead, the result will always be the value `nil`. Example 10 on page 46 gives an example of how to catch these cases.

It is better practice to use local variables wherever possible. Not only does this avoid potential name clashes between different parts of a program, but also local variable access is faster than global variable access.

Local variables are *lexically scoped*, meaning they are available only from within their current *block* starting after their point of declaration, and within any sub-blocks. Blocks are entities such as function bodies, control structure body parts, and Lua files. Local variables of the same name declared in different scopes are completely independent of one another. The following example and its accompanying call-outs exhibit the availability of local variables in various scopes:

```
-- Scoping example.
x = 1 ❶
local function y(z) ❷
    if condition then
        local x = x ❸
        block
    else
        block
        local z = x ❹
        block
    end
end
```

- ❶ *x* is a global variable. It is available anywhere a local variable *x* is not in scope.
- ❷ *y* is a local function and *z* is an implicit local argument variable. *y* is available inside itself (including its sub-blocks) and after its complete definition. *z* is available only inside *y* and its sub-blocks.
- ❸ *x* is a local variable whose initial value is the one assigned to global variable *x*. (This statement is a common idiom in Lua.) Any reassignments to local *x* do not affect global *x*. Local *x* is available only in the subsequent block below it. Outside that block (including within the *else* block), *x* refers to global *x*.
- ❹ *z* is a local variable whose initial value is global *x* (not local *x* in the *if* block). *z* is available only in the subsequent block below it (and not in the block above). Outside the lower block, *z* refers to local argument *z*.

Local Variable Attributes

Lua 5.4 introduced the ability to append the attributes “const” and “close” to local variable declarations:

```
local PI_2 <const> = math.pi / 2
local f <close> = io.open(filename)
```

<const> prevents the variable from being assigned a new value after its initial value assignment. <close> closes the variable’s value when the variable goes out of scope (e.g. via *break*, *return*, loop end, error, etc.). This *to-be-closed variable* is particularly useful for automatic resource management, as demonstrated by Example 6 on page 35. Any other attribute is considered a syntax error.

Types

Lua is a *dynamically typed* scripting language. Lua variables have no defined type, and can be assigned and reassigned any Lua value. Lua values are *first-class values*, meaning they can be assigned to variables, passed as function arguments, returned as results, and so on. The following example illustrates these concepts:

```
a = nil
a = true
a = 0
a = "string"
a = function(x, y) return x + y, x - y end
a = {1, 2, 3}
a = coroutine.create(function(x)
    coroutine.yield(x^2)
end) -- note the function that is an argument
a = io.open("filename")
```

Lua has eight basic value types: *nil*, *boolean*, *number*, *string*, *function*, *table*, *thread*, and *userdata*. Each of these types is described in the following sections.

Nil

The nil type has a single value: *nil*. It typically indicates the absence of a useful value. The default value for variables and table keys is *nil*. Assigning *nil* to a variable or table key effectively deletes it.

Booleans

Booleans have one of two values: *true* or *false*. Other than *false* and *nil*, any other value is considered to be true in a boolean sense, including the number zero, the empty string, and an empty table.

Numbers

Numbers comprise both integer numbers and floating point numbers, or *floats*. Floats are typically double-precision float-

ing point numbers, though this is configurable when compiling Lua. This book uses the term “float” in place of whatever type of float Lua is configured to use, which is not necessarily C’s single-precision float.

Numbers can be written in decimal, exponential, or hexadecimal[†] notation. Integer numbers include “0” and “-10”. Decimal floats include “-1.0” and “3.14”. Exponential floats include “6.67e-11” and “3E8”. Hexadecimal numbers include “0xFF”, “0x1P+8”, and “0X0.a”.

NOTE

Lua 5.3 and 5.4 represent numbers internally as either integers or floats and seamlessly convert between the two types as needed. The range of integers that can be represented exactly is `math.mininteger` (typically -2^{63}) to `math.maxinteger` (typically 2^{63}). Integers wrap on overflow or underflow. The function `math.type()` returns whether a given number is represented as an integer or float internally.

Lua 5.1 and 5.2 represent all numbers (including integers) internally as floats. As a result, the range of integers that can be represented exactly is typically -2^{53} to 2^{53} (for a double-precision float). Any integer outside that range loses precision.

Lua can perform arithmetic with numbers, which is described in the section “Arithmetic Operators” on page 17. Its other numeric capabilities are listed in the section “Numeric Facilities” on page 51.

Strings

Strings are immutable, arbitrary sequences of bytes. They can contain embedded zeros and have no specific encoding attached to them. Strings can be constructed using double quotes, single quotes, or brackets:

```
dq = "double-quoted string"
sq = 'single-quoted string'
ms = [[multi-line
string]]
```

[†] Except for Lua 5.1, which cannot express hexadecimal floats.

Quoted strings can contain any of the escape sequences listed in Table 2.

Bracketed strings cannot contain escape sequences, but can span multiple lines without the need for an escape sequence. If a bracketed string immediately starts with a newline, that initial newline is ignored. Similarly to block comments, bracketed string delimiters can contain an optional, equal number of ‘=’ characters between the brackets.

Table 2. *Quoted string escape sequences*

Sequence	Meaning	Sequence	Meaning
\a	Bell	\"	Double quote
\b	Backspace	\'	Single quote
\f	Form feed	\(newline)	Literal newline
\n	Newline	\z ^a	Ignore subsequent whitespace
\r	Carriage return	\ddd	Decimal byte
\t	Horizontal tab	\xhh ^a	Hexadecimal byte
\v	Vertical tab	\u{uuuu} ^b	Hexadecimal UTF-8 codepoint
\\	Literal '\'		

^a Not in Lua 5.1.

^b Not in Lua 5.1 or 5.2.

Lua can concatenate strings, and this operation is covered in the section “Other Operators” on page 21. Its facilities for creating strings, querying and transforming strings, and searching and replacing within strings are described in the section “String Facilities” on page 54.

Functions

Functions consist of both Lua functions and C functions. (Lua does not distinguish between the two.) As first-class values, functions are anonymous (they do not have names). The sections “Functions” on page 27 and “C Functions” on page 114 describe Lua and C functions, respectively.

Tables

Tables are Lua's primary data type and implement *associative arrays*. An associative array is a set of key-value pairs where keys can be any value except `nil` and `NaN`,⁴ and values can be any value except `nil`. (Therefore, if a table key is assigned `nil`, that key will no longer exist in the table.) Tables can be constructed using brace characters:

```
empty = {}  
list  = {1, 2, 3}  
dict  = {"a" = 1, ["b"] = 2, ["c"] = 3}  
mix   = {[0] = 0, 1, 2, 3, a = 1, b = 2, c = 3}
```

When keys are omitted in a table constructor, they implicitly become the integer values 1, 2, ..., *n* for the *n* values given without keys. (This kind of table with successive integer keys is considered a *list* and its values are considered *elements*.) Otherwise, keys are enclosed between brackets and are explicitly assigned values. (Both keys and values can be the results of expressions.) As a shortcut, an identifier may be used for a key. In this case, that *field* becomes a string key (e.g. the assignment “`a = 1`” is equivalent to “[`"a"`] = 1”). If the last (or only) expression in a table constructor is a function call, all of the values returned by the called function are added as trailing list elements.

NOTE

Lua's list indices start at 1, unlike C's array indices, which start at 0.

Tables are mutable and can be altered using the various operators, statements, and functions covered throughout this book. Lua always assigns, passes, and returns references to tables instead of copies of tables. Tables automatically grow in size as needed, and Lua handles all of the memory management associated with them.

Threads

Threads are separate, independent lines of execution. Instead

⁴ Not a Number is a special value for undefined numbers like 0/0.

of true multi-threading (asynchronous threads), Lua supports collaborative threads, or *coroutines*. Coroutines work together by resuming one another and then yielding to one another (a coroutine cannot be interrupted from the outside). Despite the fact that they run independently from one another, coroutines share the same global environment, and only one can be active at a time. Lua's main thread is a coroutine. The sections “Thread Facilities” and “Threading in C” on pages 68 and 133, respectively, describe Lua threads in more detail.

Userdata

Userdata act in place of C data types that cannot be represented by any other Lua value. As userdata, those C types can be treated like any other Lua value. (For example, Lua's file input and output objects are userdata.) Userdata values cannot be modified by Lua itself. The section “Push a userdata” on page 98 describes userdata in more detail.

Perform Basic Value Operations

Lua provides the means to retrieve the type of an arbitrary value, obtain the string representation of a value, and convert a string value to a number value.

`type(value)`

Returns the string type of value *value*. The returned string is either "nil", "boolean", "number", "string", "table", "function", "thread", or "userdata".

`tostring(value)`

Returns the string representation of value *value*, invoking the metamethod `__tostring()` if it exists. The section “Metatables and Metamethods” on page 32 describes metamethods.

`tonumber(value [, base])`

Returns string *value* converted to a number in base number *base*, or nil if the conversion fails. *base* must be an integer between 2 and 36, inclusive, and its default value is 10.

C API Introduction

Lua itself is just a C library. Its three header files provide the host application with a simple API for creating an embedded Lua interpreter, interacting with it, and then closing it. Example 23 demonstrates a very basic stand-alone Lua interpreter whose command line accepts only a Lua script to run.

NOTE

The C examples in this book make use of some C99-specific features, so adapting those examples on a platform without a C99-compliant compiler will likely be necessary. However, Lua itself is written in ISO (ANSI) C, and will compile without modification.

Example 23. Simple stand-alone Lua interpreter

```
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main(int argc, char **argv) {
    int status = 0;
    // Create a new embedded Lua interpreter.
    lua_State *L = luaL_newstate();
    // Load all of Lua's standard library modules.
    luaL_openlibs(L);
    // Execute the Lua script specified on the command
    // line. If there is an error, report it.
    if (argc > 1 && luaL_dofile(L, argv[1]) != LUA_OK†) {
        const char *errmsg = lua_tostring(L, -1);
        fprintf(stderr, "Lua error: %s\n", errmsg);
        status = 1;
    }
    // Close the Lua interpreter.
    lua_close(L);
    return status;
}
```

The header file *lua.h* provides Lua's basic C API. All functions and macros in that file start with the prefix “lua_”. The file *lauxlib.h* provides a higher-level API with convenience func-

† LUA_OK does not exist in Lua 5.1, which uses the constant 0 instead.

tions for common tasks that involve the basic API. All functions and macros in that file start with the prefix “luaL_”. The file *lualib.b* provides Lua’s standard library module API. Table 13 lists the contents of *lualib.b* for hosts that prefer to load only specific Lua standard library modules rather than all of them at once.

This book refers to Lua’s API functions and macros as “API functions” for the sake of simplicity.

CAUTION

Programming with Lua in C does not make programming in C any easier. Type-checking is mandatory, memory allocation errors are possible, and segmentation faults are nearly a given when passing improper arguments to Lua’s API functions. Also, any unexpected errors raised by Lua will likely cause the host program to abort. (The section “Error and Warning Handling” on page 129 describes how to avoid that unhappy scenario.)

Table 13. Standard library module API (lualib.b)

Standard Library Module Name	C Function
""	luaopen_base
LUA_BITLIBNAME ("bit32") ^a	luaopen_bit32 ^a
LUA_LOADLIBNAME ("package")	luaopen_package
LUA_MATHLIBNAME ("math")	luaopen_math
LUA_STRLIBNAME ("string")	luaopen_string
LUA_UTF8LIBNAME ("utf8") ^b	luaopen_utf8 ^b
LUA_TABLIBNAME ("table")	luaopen_table
LUA_COLIBNAME ("coroutine")	luaopen_coroutine ^c
LUA_IOLIBNAME ("io")	luaopen_io
LUA_OSLIBNAME ("os")	luaopen_os
LUA_DBLIBNAME ("debug")	luaopen_debug

^a Only in Lua 5.2.
^b Not in Lua 5.1 or 5.2.
^c Not in Lua 5.1, whose coroutine library module is included in luaopen_base.

lua_State

A C struct that represents both a thread in a Lua interpreter and the interpreter itself. Data can be shared between Lua threads but not between Lua interpreters.

TIP

Lua is fully re-entrant and can be used in multi-threaded code provided the macros `lua_lock` and `lua_unlock` are defined when compiling Lua.

lua_State *luaL_newstate();

Returns a newly created Lua interpreter, which is also that interpreter's main thread.

void luaL_openlibs(lua_State *L);

Loads all of Lua's standard library modules into Lua interpreter *L*.

`luaL_requiref(L, name, f, 1), lua_pop(L, 1);` **Lua 5.2, 5.3, 5.4**
`lua_pushcfunction(L, f), lua_pushstring(L, name),`
`lua_call(L, 1, 0);` **Lua 5.1**

Loads one of Lua's standard library modules into Lua interpreter *L*. *name* is the string name of the module to load and *f* is that module's C function. Table 13 lists Lua's standard library module names and their associated C functions.

Using this in place of `luaL_openlibs()` is useful for hosts that want control over which of Lua's standard library modules are available. For example, a host can prevent Lua code from interacting with the underlying operating system via the `os` module by simply not loading that module.

void lua_close(lua_State *L);

Destroys, garbage-collects, and frees the memory used by all values in Lua interpreter *L*. In Lua 5.4, also closes any to-be-closed variables.

Compiling Lua Programs

While a comprehensive guide to compiling Lua programs is beyond the scope of this book, the general idea is to provide a C compiler with the path to Lua's include files, and either

Concept Index

A

- anonymous functions, 14
- arithmetic operators, 17
 - invoking (stack), 106
 - metamethods for, 33
- associative arrays, 15
- attributes, 11, 100

B

- benchmarking, 80
- binary strings, 56-59
- bitwise operators, 20
 - invoking (stack), 107
 - metamethods for, 34
- block comments, 9
- blocks, 3
- booleans, 12
 - pushing (stack), 92
 - retrieving (stack), 102
- buffers, 94-96

C

- C data types, 98
- C functions, 114-122
 - arguments of,
 - handling, 116-119
 - calling, 121
 - defining, 114-116
 - environment of,
 - changing, 133
 - pushing (stack), 96
 - registering, 120
 - retrieving (stack), 104
 - upvalues of, 96, 119
 - (see also functions)
- C stack (see stack)
- calling functions, 21, 121
- classes (see object-oriented programming)
- closing metamethod, 35

- collaborative multi-threading
 - (see threads)
- command line arguments, 5, 82
- comments, 9
- compiling Lua programs, 89
- complex numbers, 125-127
- concatenation operator, 22
 - invoking (stack), 108
 - metamethod for, 36
- configuration files, 50
- control structures, 24-26
 - break from, 26
 - continue, 27
 - for, 25
 - if, 25
 - repeat, 26
 - while, 26
- coroutines (see threads)

D

- dates and times, 79-81
- directory contents, 78
- dynamic code execution, 50, 131

E

- environment variables, 4, 78
- environments, 46
 - changing, 47, 133
 - retrieving, 47, 144
 - sandboxed, 50, 131
- error handling, 48, 129-131
- exponential functions, 52
- expressions, 17-21, 23

F

- fields, 15
- files,
 - managing, 77
 - monitoring, 134-136

- files (continued)
 - opening modes, 72
 - read formats, 72
 - reading and writing, 75-77
 - size of, 75
- filesystem access, 42, 77
- first-class values, 12
- floats, 12
 - pushing (stack), 92
 - retrieving (stack), 103
- for loop, 25
- full userdata, 98
- function call operator, 21
 - invoking (stack), 121
 - metamethod for, 36
- function metamethods, 37
- functions, 14, 27-31
 - anonymous, 14
 - calling, 21
 - default argument values, 28
 - defining, 28
 - named arguments, 29
 - returning values from, 29, 31
 - tail calls, 29
 - upvalues of, 28
 - variable arguments in, 30 (see also C functions)

G

- garbage collection, 82, 99, 143
 - metamethod for, 99
- global environment, 46
- global variables, 10, 46, 106
- goto, 27

H

- host application, 1, 3, 87

I

- identifiers, 9

- if conditional, 25
- input and output, 72-78
 - files, with, 75-77
 - processes, with, 78
 - simple, 73
- integers, 12, 54
 - pushing (stack), 92
 - retrieving (stack), 103
- iterating over tables, 64, 111
 - metamethods for, 37
- iterators, 26

L

- labels, 27
- length operator, 22
 - invoking (stack), 112
 - metamethod of, 37
- lexical scoping, 10
- light userdata, 98
- line comments, 9
- lists,
 - concatenating elements
 - of, 67
 - creating, 15
 - definition of, 4
 - elements in, 15
 - indices of, 15, 64
 - iterating over, 64
 - length of, 22
 - manipulating, 66
 - permutations of, 69
 - unpacking, 66 (see also tables)
- loading dynamic code, 50, 131
- local variables, 10
- locales, 81
- logarithmic functions, 52
- logical operators, 19
- Lua,
 - downloading, 2
 - editing, 2
 - file extension for, 9
 - overview of, 1

Lua interpreter,
 closing, 89
 creating, 89
 stand-alone, 1, 5, 87
LuaJIT, 1

M

main thread, 89
mathematical functions, 51
memory management, 82,
 143
metamethods, 32-38
 arithmetic, 33
 bitwise, 34
 bypassing, 38
 concatenation, 36
 function, 37
 function call, 36
 garbage collection, 99
 invoking in C, 124
 length, 37
 relational, 33
 table index, 36
 table iteration, 37
metatables, 32, 122-124
 assigning, 32, 123
 creating in C, 123
 retrieving, 32, 123
modules, 42-46, 124-129
 creating, 45, 124, 128
 loading, 5, 43, 89, 128
 unloading, 44
multi-line comments, 9
multi-line strings, 13
multi-threading (see threads)

N

nil, 12
 pushing (stack), 92
numbers, 12
 pushing (stack), 92
 retrieving (stack), 103
numeric for loop, 25

O

object-oriented
 programming, 38-42
 classes, defining, 38-41
 classes, invoking, 41
 inheritance, 40
 methods, defining, 41
 multiple inheritance, 41
opening files (see files)
operator overloading, 32
operators, 17-23
 arithmetic, 17
 bitwise, 20
 concatenation, 22
 function call, 21
 length, 22
 logical, 19
 overloading, 32
 precedence of, 17
 relational, 18
 table index, 22
 ternary, 20

P

Parsing Expression
 Grammars, 42
patterns, 60-62
prime number
 generation, 25
processes, 78
protected calls, 48, 129

R

random numbers, 53
read-only tables, 36
reading from files (see files)
reference system, 113
registry, 113
regular expressions, 60
relational operators, 18
 invoking (stack), 107
 metamethods for, 33
repeat until loop, 26

- reserved words, 9
- return statement, 29, 31
- running dynamic code, 50, 131

S

- sandboxed environments, 50, 131
- scopes, 10
- Sieve of Eratosthenes, 25
- sockets, 42
- stack, 90-114
 - arithmetic operations, 106
 - bitwise operations, 107
 - element operations, 105
 - global variable
 - operations, 106
 - indices, 91
 - length operations, 112
 - overflow, 91
 - popping values, 101
 - pushing values, 92-100
 - querying values, 101
 - reference operations, 113
 - relational operations, 107
 - retrieving values, 102-105
 - size, 91, 120
 - string operations, 108
 - table operations, 109-112
- stand-alone Lua
 - interpreter, 1, 5, 87
- statements, 23-27
- strings,
 - binary, working
 - with, 56-59
 - buffers, 94-96
 - concatenating, 22
 - creating, 13, 54-56, 67
 - escape sequences in, 14
 - indices of, 54
 - length of, 22
 - patterns, 60-62
 - pushing (stack), 93-96

- querying, 59
- retrieving (stack), 103
- searching and replacing
 - in, 60-63
- stack operations on, 108
- transforming, 59
- UTF-8, working with, 63

T

- table assignment
 - statements, 24
 - invoking (stack), 110
 - metamethods for, 36
- table index operators, 22
 - invoking (stack), 109
 - metamethods for, 36
- tables,
 - assigning values in, 24, 110
 - creating, 15, 97
 - environments, 46
 - fields in, 15
 - indexing, 22, 109
 - iterating over, 64, 111
 - pretty-printing, 67
 - pushing (stack), 97
 - read-only, 36
 - registry, 113
 - stack operations
 - on, 109-112
 - (see also lists)
- tail calls, 29
- Textadept, 2
- the stack (see stack)
- threads, 15, 67-72, 133-142
 - calling yielding
 - functions, 140-142
 - closing, 71, 139
 - creating, 70, 136
 - main thread, 89
 - procedure for, 67, 133
 - pushing (stack), 98
 - querying, 71, 139
 - resuming, 70, 137

- retrieving (stack), 104
- starting, 70, 137
- transferring data between
 - in C, 139
- yielding, 70, 138
- times and dates, 79-81
- to-be-closed variables, 11, 35
 - metamethod for, 35
- trigonometric functions, 52
- types, 12-16
 - boolean, 12
 - converting between, 16, 143
 - determining, 16, 101
 - function, 14
 - nil, 12
 - number, 12
 - string, 13
 - table, 15
 - thread, 15
 - userdata, 16

U

- unit testing, 42
- upvalues, 4, 28, 96, 119

- URL parser, 60
- user values, 98
- userdata, 16, 98-100
 - full userdata, 98
 - light userdata, 98
 - pushing (stack), 98-100
 - retrieving (stack), 105
- UTF-8 strings, 63

V

- values (see types)
- variables,
 - assigning, 23
 - attributes for, 11, 100
 - global, 10, 46, 106
 - local, 10
 - multiple assignment, 23
 - scope of, 10
 - swapping values of, 24
 - to-be-closed, 11, 35
- vectors, 39

W

- warnings, 6, 48, 119, 129
- while loop, 26
- writing to files (see files)