mEm Inc

3ND EDITION

Lua Programming

The Ultimate Beginner's Guide to Learn Lua Step by Step

Claudia Alves & Alexander Aronowitz

Lua Programming
The Ultimate Beginner's Guide to Learn Lua Step by Step

Third edition

2021

By
Claudia Alves
& Alexander Aronowitz

TABLE OF CONTENTS

Introduction
The audience
About the third edition
Other resources
Certain Typographical Conventions
Running examples
Acknowledgments

PART I
Language
Chapter 1. Getting Started
1.1. Blocks
1.2. Some lexical conventions
1.3. Global variables
1.4. Separate interpreter
Exercises

Chapter 2. Types and values
2.1. Nil
2.2. Boolean
2.3. Numbers
2.4. Strings
Literals
Long lines
Type conversions
2.5. Tables
2.6. Functions
2.7. userdata and threads
Exercises
Chapter 3. Expressions
3.1. Arithmetic Operators
3.2. Comparison operators
3.3. Logical operators
3.4. Concatenation
3.5. Length operator
3.6. Operator Priorities
3.7. Table constructors
Exercises
Chapter 4. Operators
4.1. Assignment operators
4.2. Local variables and blocks
4.3. Control structures
if then else
while
repeat
Numeric for statement
General for statement

4.4. break, return and goto
Exercises
Chapter 5. Functions
5.1. Multiple Results
5.2. Functions with a variable number of arguments
5.3. Named arguments
Exercises
Chapter 6. More about functions
6.1. Closures
6.2. Non-global functions
6.3. Optimization of tail calls
Exercises
Chapter 7. Iterators and the generic for
7.1. Iterators and Closures
7.2. Generic for semantics
7.3. Stateless iterators
7.4. Complex State Iterators
7.5. True iterators
Exercises
Chapter 8. Compilation, Execution, and Errors
8.1. Compilation
8.2. Precompiled Code
8.3. C code
8.4. Errors
8.5. Error and Exception Handling
8.6. Error messages and call stack
Exercises

Chapter 9. Coroutines
9.1. Basics of coroutines
9.2. Channels and Filters
9.3. Coroutines as iterators
9.4. Non-displacing multi-threading
Exercises
Chapter 10. Completed Examples
10.1. The problem of eight queens
10.2. The most common words
10.3. Markov chain
Exercises
PART II
Tables and Objects
Chapter 11. Data Structures
11.1. Arrays
11.2. Matrices and multidimensional arrays
11.3. Linked Lists
11.4. Queues and double queues
11.5. Sets and Sets
11.6. Line buffers
11.7. Counts
Exercises
Chapter 12. Data Files and Persistence
12.1. Data files
12.2. Serialization
Saving tables without loops
Saving tables with loops
Exercises

Chapter 13. Metatables and Metamethods
13.1. Arithmetic metamethods
13.2. Comparison Methods
13.3. Library Metamethods
13.4. Methods for accessing the table
__Index metamethod
__Newindex metamethod
Default tables
Tracking access to a table
Read-only tables
Exercises
Chapter 14. Environment
14.1. Global variables with dynamic names
14.2. Descriptions of global variables
14.3. Non-global environments
14.4. Using _ENV
14.5. _ENV and load
Exercises
Chapter 15. Modules and Packages
15.1. Require function
Renaming a module
Search along the path
File crawlers
15.2. The Standard Approach for Writing Lua Modules
15.3. Using environments
15.4. Submodules and Packages
Exercises
Chapter 16. Object Oriented programming
16.1. Classes

16.2. Inheritance
16.3. Multiple inheritance
16.4. Hiding
16.5. Single Method Approach
Exercises
Chapter 17. Weak Tables and Finalizers
17.1. Weak tables
17.2. Functions with caching
17.3. Object Attributes
17.4. Again tables with default values
17.5. Ephemeral tables
17.6. Finalizers
Exercises
PART III
Standard Libraries
Chapter 18. Math Library
Exercises
Chapter 19. Library for Bitwise Operations
Exercises
Chapter 20. Library for Working with Tables
20.1. Insert and remove functions
20.2. Sorting
20.3. Concatenation
Exercises
Chapter 21. Library for working with strings
21.1. Basic functions for working with strings
21.2. Functions for working with templates
String.find function

String.match function
String.gsub function
String.gmatch function
21.3. Templates
21.4. Grips
21.5. Substitutions
URL encoding
Replacing tabs
21.6. Tricky tricks
21.7. Unicode
Exercises
Chapter 22. Library I / O
22.1. Simple I / O Model
22.2. Full I / O model
A small trick to increase performance
Binaries
22.3. Other operations with files
Exercises
Chapter 23. Library of operating room functions systems
23.1. Date and time
23.2. Other system calls
Exercises
Chapter 24. Debug Library
24.1. Accessibility (introspection)
Accessing Local Variables
Access to non-local variables
Access to other coroutines
24.2. Hooks
24.3. Profiling
Exercises

PART IV
WITH API
Chapter 25. Overview of the C API
25.1. First example
25.2. Stack
Putting items on the stack
Referring to items
Other stack operations
25.3. Error handling in the C API
Handling errors in the application code
Error handling in the library code
Exercises
Chapter 26. Extending Your Application
26.1. Basics
26.2. Working with tables
26.3. Lua function calls
26.4. Generalized function call
Exercises
Chapter 27. Calling C from Lua
27.1. Functions on C
27.2. Continuations
27.3. C modules
Exercises
Chapter 28. Techniques for writing functions in C
28.1. Working with arrays
28.2. Working with strings
28.3. Saving state in functions on C
Register
Function related values
Function related values used
several functions

Exercises
Chapter 29. User-Defined Types in C
29.1. User data (userdata)
29.2. Metatables
29.3. Object Oriented Access
29.4. Access as an ordinary array
29.5. Light objects of type userdata (light userdata)
Exercises
Chapter 30. Resource Management
30.1. Directory iterator
30.2. XML Parser
Exercises
Chapter 31. Threads and States
31.1. Numerous threads
31.2. Lua States
Exercises
Chapter 32. Memory Management
32.1. Memory allocation function
32.2. Garbage collector
Garbage collector API
Exercises

INTRODUCTION

When Waldemar, Louis, and I started developing Lua in 1993, we could hardly imagine that Lua would spread like that. On- started as home language for two specific projects, now Lua is widely used in all areas one can get
benefits from simple, extensible, portable and efficient scripting language such as embedded systems, mobile devices swarms and, of course, games. We designed Lua from the beginning to integrate with software software written in C / C ++ and other common strange languages. There are many benefits to this integration. Lua is a tiny and simple language, partly because it doesn't tries to do what C is already good at, such as speed,
low-level operations and interaction with third-party programs parties. Lua relies on C for these tasks. Lua offers something for which C is not good enough: sufficient distance from the hardware go support, dynamic structures, no redundancy and ease of testing and debugging. For these purposes, Lua has safe environment, automatic memory management and good possibilities for working with strings and other types
resizable data. Some of Lua's strength comes from its libraries. And this is no coincidence. In the end Finally, one of the main strengths of Lua is extensibility. Many language features contribute to this. Dynamic typing tion provides a large degree of polymorphism. Automatic memory management simplifies interfaces because there is no need to the ability to decide who is responsible for allocating and freeing memory or how to handle overflows. Higher-order functions and anonymous functions allow a high degree of parameterization, making functions more versatile.

More than an extensible language, Lua is a “ *glue vayushim ”(glue) language* . Lua supports a component-based approach to software development when we create an application
by gluing together existing high-level components. These components are written in a compiled language with a static typing such as C / C ++; Lua is the glue we use use to arrange and connect these components. Usually components (or objects) represent more specific low- tier entities (such as widgets and data structures), which which hardly change during the development of the program and which take up the bulk of the final program execution time. Lua gives the final form to the application, which is most likely changes a lot during the life of a given software product. However, unlike other "glue" technologies, Lua is is a complete programming language. Therefore we can use Lua not only to "glue" components, but also to adaptation and customization of these components, as well as to create a floor new components. Of course, Lua isn't the only scripting

language. Exists other languages you can use for roughly the same goals. Nevertheless, Lua provides a whole set of possibilities, which make him the best choice for many tasks and gives him your unique profile:

• *Extensibility* . Lua's extensibility is so great that many consider Lua not as a language, but as a set for DSL structures (domain-specific language, a language created for specific area, application). We developed Lua from the beginning so that it is extensible both through code on Lua as well as through C code. How the Lua proof implementsmost of its basic functionality via external library. Interaction with C / C ++ really simple and Lua has been successfully integrated with many others languages such as Fortran, Java, Smalltalk, Ada, C #, and even with scripting languages like Perl and Python.

• *Simplicity* . Lua is a simple and small language. It is founded on a small number of concepts. This simplicity makes it easier to study nie. Lua contributes to its very small size. Floor- distribution kit (source code, manual, binaries ly for some platforms) is quietly placed on one floppy disk.

• *Efficiency* . Lua has a very efficient implementation tions. Independent tests show that Lua is one of the most most fast languages among scripting languages.

• *Portability* . When we talk about portability, we say Rome on running Lua on all platforms you are only talking about heard: all versions of Unix and Windows, PlayStation, Xbox, Mac OS X and iOS, Android, Kindle Fire, NOOK, Haiku, QUALCOMM Brew, big servers from IBM, RISC OS, Symbian OS, process Sora Rabbit, Raspberry Pi, Arduino and more. Original the code for each of these platforms is almost the same. Lua does not use conditional compilation to adapt its code for different machines, instead it keeps standard ANSI (ISO) C. Thus, you usually do not you need to adapt it to the new environment: if you have a com- piler with ANSI C, then you just need to compile Lua.

The audience

Lua users generally fall into one of three broad groups: those who use Lua already built into the application, those who use use Lua separately from any application (standalone), and those who use Lua and C together. Many people use Lua built into some application, like Adobe Lightroom, Nmap, or World of Warcraft. These pri- applications use the Lua-C API to register new functions,

creating new types and changing the behavior of some operations language by configuring Lua for its area. Often users are whom applications don't even know that Lua is an independent language, adapted for the given area. For example, many developers Lightroom plugins are unaware of other uses of this language; Nmap users generally view Lua as scripting Nmap language; World of Warcraft players can view Lua as a language exclusively for this game.

Lua is also useful as just an independent language, not only for word processing and one-time small programs, but also for a variety of medium to large sized projects. For the In this way, the main functionality of Lua comes from its libraries. Standard libraries, for example, provide basic new functionality for working with templates and other functions for working with strings. As Lua improves its support libraries, a large number of external packages appeared. Lua Rocks, a system for building and managing modules for Lua, now has over 150 packages. Finally, there are programmers who use Lua as a library flow for C. Such people write more in C than in Lua, although they need requires a good understanding of Lua to create interfaces that are simple, easy to use and well integrated tied with the tongue.

This book can be useful to all of these people. The first part covers the tongue itself, showing how the whole its potential. We focus on various language constructs and use numerous examples and exercises to show
know how to use them for practical tasks. Some chap- this part covers basic concepts such as managing structures, while the rest of the chapters cover more advanced tricky topics like iterators and coroutines.

The second part is completely devoted to tables, the only structure data tour in Lua. The chapters in this part discuss data structures, persistence, packages and object-oriented programming. It is there that we will show the full power of language. The third part introduces the standard libraries. This part especially useful for those who use Lua on their own language, although many applications include part or all standard libraries. In this part, each library is dedicated separate chapter: math library, bitwise library, library for working with tables, library for working with strings mi, I / O library, operating system library and debug library. Finally, the last part of the book covers the API between Lua and C.

This section is markedly different from the rest of the book. In this part we will be programming in C, not Lua. For some, this part may be uninteresting, but for someone - on the contrary, the most useful part of the book.

Running examples

You will need a Lua interpreter to run the examples from this books. Ideally, you should use Lua 5.2, however most in the examples will work on Lua 5.1 without any changes. The Lua site (http://www.lua.org) stores all the source code for interpreter. If you have a C compiler and know how compile the C code on your computer, you better ask try to install Lua from source; it's really easy. The *Lua Binaries* site (look for luabinaries) already offers compiled native interpreters for all major platforms. If you using Linux or another UNIX-like system, you can check the repository of your distribution; many distributions already offer ready-made packages with Lua. For Windows, a good choice is rum is *Lua for Windows* (look for luaforwindows), which is a convenient set for working with Lua. It includes interpreting torus, integrated editor and many libraries.
If you are using Lua embedded in an application like WoW or Nmap, then you may need a manual for this. placement (or the help of a "local guru") in order to understand, how to run your programs. Nevertheless, Lua remains all the same the same language; most of the examples we will look at in this book are applicable regardless of how you use Lua. But I recommend that you start learning Lua with an interpreter to run you examples.

Part I
I am the language

CHAPTER 1

Begin

Continuing the tradition, our first Lua program simply printed em "Hello World" : print ("Hello World") If you are using a separate Lua interpreter, then all you need is you need to run your first program - this is to run the interpreter tator - usually called lua or lua5.2 - with the name of the text file containing your program. If you saved the above the above program in hello.lua file , then you should run following command:

% lua hello.lua

As a more complex example, our next program defines There is a function for calculating the factorial of a given number, asking gives the user a number and prints its factorial:

- defines a factorial function

```
function fact (n)
if n == 0 then
return 1
else
return n * fact (n-1)
end
end
print ("enter a number:")
a = io.read ("* n") - reads a number
print (fact (a))
```

1.1. Blocks

Every piece of code that Lua executes, such as a file or from- a smart string in interactive mode is called *a* chunk. A block is simply a sequence of commands (or statements). Lua does not need a separator between consecutive operators, but you can use semicolon if you like. I personally use I use a semicolon only to separate statements written in one line. Line splitting does not play any role in the syntax system Lua; so, the following four blocks are valid and equivalent:

```
a = 1
b = a * 2
a = 1;
b = a * 2;
```

a = 1; b = a * 2
a = 1 b = a * 2 - ugly, but valid

A block can consist of just one statement, as in the example "Hello World", or consist of a set of operators and definitions functions (which are actually just assignments, as we will see later), as in the factorial example. Block can be as great as you want. Since Lua is also used as language for describing data, blocks of several megabytes are not a rarity. The Lua interpreter does not have any problems with bot with large blocks.
Instead of writing your programs to a file, you can run the interpreter interactively. If you run- those lua without arguments, then you will see its prompt for input:

% lua
Lua 5.2 Copyright (C) 1994-2012 Lua.org, PUC-Rio
>

Accordingly, each command that you enter (like, for example measure, print “Hello World”) is executed immediately after how you enter it. To exit the interpreter, just type end-of-file character (ctrl-D in UNI, ctrl-Z in Windows) or call the exit function from the operating system library - you need type os.exit () .
In interactive mode, Lua usually interprets each the line you enter as a separate block. However, if he is detects that the line is not a complete block, then it waits continue typing until a complete block is obtained. This way you can enter multi-line definitions, so as a factiorial function , directly interactively. One- but it is usually more convenient to place such definitions to a file and then call Lua to execute that file. You can use the –i option to force Lua switch to interactive mode after executing the given block ka:

% lua -i prog

A command like this will execute the block in the prog file and then go into interactive mode. This is especially useful for debugging and manual th testing. At the end of this chapter, we will look at other options. command line for the Lua interpreter. Another way to trigger blocks is the dofile function , which paradise executes the file immediately. For example, let's say you have lib1.lua file with the following code:

function norm (x, y)
return (x ^ 2 + y ^ 2) ^ 0.5
end

function twice (x)
return 2 * x
end

Then interactively you can type

> dofile (“lib1.lua”) - load your library
> n = norm (3.4, 1.0)
> print (twice (n)) -> 7.0880180586677

The dofile function is also useful when you are testing a piece of code. You can work with two windows: one contains the text editor with your program (for example, in the prog.lua file), and in other gom is a console running the Lua interpreter in the inter- active mode. After you have saved the changes to your program, you do dofile (“prog.lua”) in the console to load ki new code; then you can start using the new code, calling functions and printing the results.

1.2. Some lexical agreements

Identifiers (or names) in Lua are strings from Latin their letters, numbers and underscores that do not start with a number;

eg:

ij i10 _ij
aSomewhatLongName _INPUT

You are better off avoiding identifiers consisting of underscores vana followed by capital Latin letters (for example, _VERSION); they are reserved for special purposes in Lua. I usually use id _ (single underscore)
for dummy variables. In older versions of Lua, the concept of what a letter is depended on from the locale. However, these letters make your program un-suitable to run on systems that do not support this locale. Therefore, Lua 5.2 considers only letters as letters. from the following ranges: AZ and az . The following words are reserved, you cannot use them as identifiers:

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

until while

Lua is case sensitive: **and is** a reserved word, however, And and AND are two different identifiers. The comment starts with two minus signs (-) and continues with- until the end of the line. Lua also supports block comments, which starts with - [[and goes to the next]] 1 . Standard the way to comment out a piece of code is to put it between
- [[and -]] as shown below:

```
- [[
print (10) - no action (commented out)
-]]
```

To make this code active again, just add one minus to the first line:

```
--- [[
print (10) -> 10
-]]
```

In the first example - [[in the first line starts a block com- mentary, and the double minus in the last line is also in this comment. In the second example --- [starts the usual single line comment, so the first and last lines
become regular independent comments. In this case print is outside of comments.

1.3. Global Variables

Global variables do not need descriptions; you just use them you whine. It is not an error to refer to uninitialized variable; you just get nil as a result- that:

```
print (b) -> nil
b = 10
print (b) -> 10
```

If you assign nil to a global variable, Lua will itself as if this variable has never been used:

```
b = nil
print (b) -> nil
```

After this assignment, Lua may eventually regain its password. the space occupied by this variable.

1.4. Separate interpreter

Stand-alone interpreter (also called lua.c in relation to the name of its source file or just lua by name executable file) is a small program that allows direct use of Lua. This section presents her basic options.
When the interpreter loads a file, it skips the first string if it starts with '#' character . This allows you to use make Lua a scripting interpreter on UNIX systems. If you start your script with something like

#! / usr / local / bin / lua

(assuming the interpreter is in / usr / local / bin) or

#! / usr / bin / env lua ,

then you can directly run your script without explicitly start the Lua interpreter. The interpreter is called like this:

lua [options] [script [args]]

All parameters are optional. As we have seen, when we start If lua has no arguments, it goes into interactive mode.
The -e option allows you to directly specify the code directly in the command-line like below:

% lua -e "print (math.sin (12))" -> -0.53657291800043

(UNIX requires double quotes so that the command interpreter the torus (shell) did not parse parentheses).
The –l option loads the library. As we have seen previously, -i pe- puts the interpreter into interactive mode after processing the OS arguments. So the next call will load the library lioteku lib , then execute the assignment x = 10 and finally pass into interactive mode.

% lua -i -llib -e "x = 10"

Interactively, you can print the value of the expression by simply typing a line that starts with an equal sign for
followed by the expression:

> = math.sin (3) -> 0.14112000805987
> a = 30
> = a -> 30

This feature allows Lua to be used as a calculator. Before executing its arguments, the interpreter looks for re- environment variable named LUA_INIT_5_2 or, if such a change no, no, LUA_INIT . If one of these variables is present and its has the form *@filename* , the interpreter launches this file. If LUA_INIT_5_2 (or LUA_INIT) is defined but not start- is started with the '@' character , then the interpreter assumes that it contains executable Lua code and executes it. LUA_INIT gives huge the ability to configure the interpreter, since when configuration, all the power of Lua is available to us. We can download packages, change the current path, define your own functions change, rename or delete functions, etc.
The script can get its arguments in a global variable
arg . If we have a call like % lua script abc , then the interpreter creates arg table with all command line arguments before by executing the script. The script name is located at index 0, per- the first argument (in the example it is "a") is located at index 1, and so on. The preceding options are arranged in negative indices, as they appear before the script name. For example, ra- look at the following call:

% lua -e "sin = math.sin" script ab

The interpreter collects arguments as follows:

arg [-3] = "lua"
arg [-2] = "-e"
arg [-1] = "sin = math.sin"
arg [0] = "script"
arg [1] = "a"
arg [2] = "b"

Most often the script uses only positive indices (in the example, these are arg [1] and arg [2]).
Since Lua 5.1 the script can also receive its arguments using the expression ... (three dots). In the main part of the script, this is expression gives all the arguments of the script (we will discuss similar expressions (see section 5.2).

Exercises

Exercise 1.1 . Run the factorial example. What will happen with your program if you enter a negative number? Modify the example to avoid this problem.

Exercise 1.2 . Run the twice example by loading once file using the –l option , and another time using dofile . what faster?

Exercise 1.3 . Can you name another language using- (-) for comments?

Exercise 1.4 . Which of the following lines are valid by our identifiers?
___ _end End end until? nil NULL

Exercise 1.5 . Write a simple script that prints your name without knowing it in advance.

CHAPTER 2
Types and values

Lua is a dynamically typed language. The language has no definitions of ty-pov,

each value carries its own type.
There are eight basic types in Lua: *nil* , *boolean* , *number* , *string* , *userdata* , *function* , *thread* and *table* . The type function returns the type for any passed value:

```
print (type (“Hello world”))
-> string
print (type (10.4 * 3))
-> number
print (type (print))
-> function
print (type (type))
-> function
print (type (true))
-> boolean
print (type (nil))
-> nil
print (type (type (X)))
-> string
```

The last line will always return **string** regardless of the value Niya the X , as the result of a function type is always a string.
Variables have no predefined types; any variable can contain values of any type:

```
print (type (a)) -> nil ('a' not yet defined)
a = 10
print (type (a)) -> number
a = “a string !!”
print (type (a)) -> string
a = print
-- Yes it is possible!
a (type (a))
-> function
```

Notice the last two lines: functions are first class values in Lua; they can be manipulated like any other values. (More on this in Chapter 6.) Usually when you use the same variable for value different types, you get disgusting code. However, sometimes judicious use of this opportunity is beneficial, for example using **nil** to distinguish normal return value from any error.

2.1. Nil

Nil is a type with only one value, **nil** , the main which is different from all other

values. Lua
uses nil to indicate a missing value. Like us already seen, globals default to nil before its first assignment, you can also assign nil a global variable to delete it.

2.2. Boolean (boolean values)

The boolean type has two values, ***true*** and ***false*** , which serve to pre- setting traditional logical values. However, these values do not monopolize all conditional values: in Lua, any value can represent a condition. Relevant checks (checking conditions in various control structures) interpret both **nil** and **false** as false and all other values as valid muddy. In particular, Lua treats zero and an empty string as true. in logical conditions. Throughout the book, false will mean **nil.** and **false** . In the case when it is about boolean values, there will be explicitly set to **false** .

2.3. Numbers

The number type represents floating point values specified with double precision. Lua does not have a built-in integer type.
Some fear that even simple operations such as increase by one (increment) and comparison, may be incorrect work with floating point numbers. However, in fact it isnot this way. Almost all platforms now support the standard IEEE 754 for floating point representation. According to this standard, the only possible source of errors is There is a case where the number cannot be accurately represented. Opera- walkie-talkie rounds its result only if the result cannot be accurately represented as the corresponding float value point. Any operation whose result can be accurately predicted put, will have the exact meaning. In fact, any integer up to 2^{53} (approximately 10^{16}) has an exact floating point representation with double precision. When you use a float value double-precision dot to represent integers, no rounding errors unless the value is greater than 2^{53} in absolute value . In particular, Lua is capable of representing any 32-bit integer values without rounding problems. Of course, fractional numbers will have rounding problems. This the situation is no different than when you have paper and a pen. If we want to write 1/7 in decimal form, then we must somewhere us stop. If we use ten digits to represent number, then 1/7 becomes 0.142857142 . If we calculate 1/7 * 7 s ten digits, we get 0.999999994 , which is different from 1. Moreover, numbers that have a finite representation in the form

decimal fractions can have an infinite representation in the form binary fractions. So, 12.7-20 + 7.3 is not zero, since both numbers 12.7 and 7.3 do not have an exact binary representation (see
Figure 2.3). Before we continue, remember that integers have an exact representation and therefore have no rounding errors. Most modern CPUs perform floating operations. dot as fast (or even faster) than with integers. However, it is easy to compile Lua so that for numeric values a different type was used, for example long integers single-precision floating-point values or numbers. It especially useful for platforms without hardware support for numbers with floating point such as embedded systems. For de- For this, refer to the luaconf.h file in the Lua source files. We can write numbers, if necessary, specifying a fraction part and decimal degree. Examples of valid numeric constants are:

4 0.4 4.57e-3 0.3e12 5E + 20

Moreover, we can also use hexadecimal constants starting at 0x . Since Lua 5.2 hexadecimal constants can also have a fractional part and a binary power (ne- 'p' or 'P' is used before the degree), as in the following examples:

0xff (255) 0x1A3 (419) 0x0.2 (0.125) 0x1p-1 (0.5)
0xa.bp2 (42.75)

(We've added a decimal representation for each constant.)

2.4. Strings

Lua strings have the usual meaning: a sequence of characters fishing. Lua supports all 8-bit characters, and strings can be keep characters with any codes, including zeros. It means that you can store any binary data as strings. you also you can store unicode strings in any representation (UTF-8, UTF-16, etc.). The standard library that comes with Lua is does not contain built-in support for these views. However less, you may well be working with UTF-8 strings, which we will consider Rome in section 21.7. Lua strings are immutable values. You can not change the character inside the string, as you can in C; instead of this you create a new line with the desired changes as shown in following example:

a = “one string”
b = string.gsub (a, “one”, “another”) - change part of the string
print (a) -> one string

print (b) -> another string

Lua strings are subject to automatic memory management, just like other Lua objects (tables, functions, etc.). This is a sign cheat that you don't have to worry about allocating and freeing lines; Lua will do it for you. The string can be one character or a whole book. Programs that work with lines of 100K or 10M characters are not uncommon in Lua. You can get the length of a string using as a prefix operator '#' (called length operator):

a = “hello”
print (#a) -> 5
print (# "good \ 0bye") -> 8

Literals

We can put strings inside single or double quotes- check:
a = “a line”
b = 'another line'

These types of records are equivalent; the only difference is what is inside a string, limited to one type of quotation marks, you can directly insert quotation marks of a different type. Usually most programmers use quotation marks of one type for the same string type. For example, a library that works with XML, can use single quotes for strings, containing XML fragments, since these fragments often contain
double quotes. Lua strings can contain the following escape sequence nos:

\ a
bell
\ b
Back space
\ f
page translation (form feed)
\ n
newline
\ r
carriage return
\ t
tab (horizontal tab)
\ v
vertical tab
****

backslash
\ "
double quote
\ '
single quote

The following example illustrates their use:

```
> print (“one line \ nnext line \ n \” in quotes \ ”, 'in quotes'”)
one line
next line
“In quotes”, 'in quotes'
> print ('a backslash inside quotes: \' \\\ '')
a backslash inside quotes: '\'
> print (“a simpler way: '\\'”)
a simpler way: '\'
```

We can specify a character in a string using its numeric value using constructions like \ *ddd* and \ x \ *hh* , where *ddd* is a sequence of no more than three decimal digits, and *hh* is a sequence of exactly two hexadecimal digits. As complex example two lines “alo \ n123 \” ” and '\ 97lo \ 10 \ 04923”' have the same meaning on a system using ASCII: 97 is the ASCII code for 'a' , 10 is the code for the line feed character, and 49 is the code for the digit '1' (in this example we should write the value 49 using three decimal digits \ 049 , because it is followed by another number; otherwise Lua interpreted it as code 492). We we can also write the same string as ' \ x61 \ x6c \ x6f \ x0a \ x31 \ x32 \ x33 \ x22 ' , representing each character in its sixteen-normal value.

Long lines

We can delimit character strings with double square brackets, as we did with the comments. Line in this form can span many lines, and control sequences the numbers on these lines will not be interpreted. Moreover, this the form ignores the first character of the line if it is a jump character next line. This form is especially useful for writing lines, containing large snippets of code as shown below:

```
page = [[
<html>
<head>
<title> An HTML Page </title>
</head>
```

```
<body>
<a href=32http://www.lua.org> Lua </a>
</body>
</html>
]]
write (page)
```

Sometimes you may want to put something like
a = b [c [i]] (note the]] in this code) or you can want to put in a line a piece of code where some piece is already commented out. To deal with such cases, you can use place any number of equal signs between two opening in square brackets, for example [=== [. After that the line will terminate only on a pair of closing square brackets with the same the most equal signs ( ] ===] for our example). The scanner will ignore pairs of brackets with a different number of values. kov equality. By choosing a suitable number of equal signs You can wrap any fragment in a string. The same is true for comments. For example, if you start- those long comment with - [= [, then it will continue all the way before] =] . This feature allows you to comment out any fragment a code snippet containing already commented out fragments.
Long lines are very handy for including text in your code, but you shouldn't use them for non-text strings. Although the lines in Lua can contain any characters, this is not a good idea - use these symbols in your code: you may run into problems with your text editor; moreover, lines of the form "\ R \ n" can become "\ n" . Therefore, to represent the arbitrary binary data, it is better to use control software sequences starting with a \ , such as \ x13 \ x01 \ xA1 \ xBB ". However, this presents a problem for long lines because of the resulting length. For situations like this, Lua 5.2 offers a control after- the \ z sequence : it skips all characters in the string up to the first non-whitespace character. The following example illustrates its use use:

```
data = "\ x00 \ x01 \ x02 \ x03 \ x04 \ x05 \ x06 \ x07 \ z
\ x08 \ x09 \ x0A \ x0B \ x0C \ x0D \ x0E \ x0F "
```

\ Z at the end of the first line skips the following
end of line and indentation of next line so that byte
\ x07 immediately followed by a byte \ x08 in the resulting row.

Type casts

Lua provides automatic conversion of values between

strings and numbers at run time. Any numeric operation applied to a string, tries to convert the string to a number:

print (“10” + 1) -> 11
print (“10 + 1”) -> 10 + 1
print (“- 5.3e-10” * ”2”) -> -1.06e-09
print (“hello” + 1) - ERROR (cannot convert “hello”)

Lua applies similar conversions not only in arithmetic operators, but also in other places where the expected number, for example for the math.sin argument . Similarly, when Lua expects to receive a string, but receives a number, it converts the number to string:

print (10 .. 20) -> 1020

(The .. operator is used in Lua to concatenate strings. When you use it write immediately after the number, then you must separate them from each a friend with a space; otherwise Lua will assume that the first point is decimal point of the number.) Today we are not sure if these automatic conversions types were a good idea in Lua design. It is generally best on do not count them. They are handy in some places; but adding- There are complexities in both the language and the programs that use them. After all, strings and numbers are different types despite everything these transformations. A comparison like 10 = "10" gives a false value, because 10 is a number and “10” is a string. If you need to explicitly convert a string to a number, then you can use the tonumber function , which returns **nil** if string ka does not contain a number:

line = io.read ()
- read the line
n = tonumber (line)
- try to translate it into a number
if n == nil then
error (line .. “is not a valid number”)
else
print (n * 2)
end

To convert a number to a string, you can use the function tion tostring or concatenate the number with an empty string:

print (tostring (10) == “10”)
-> true
print (10 .. “” == “10”)

-> true
These transformations always work.

2.5. Tables

The table type corresponds to an associative array. Associative a ny array is an array that can be indexed not only numbers, but also strings or any other value from the language, except for **nil** .
Tables are the main (actually the only) place a hanism for structuring data in Lua, and very powerful. We use tables to represent regular arrays, sets, records and other data structures simple, homogeneous and in an efficient way. Lua also uses tables to pre- placing packages and objects. When we write io.read , we think- This is about the " read function from the io module ". For Lua, this expression means "Take the value from the io table by the read key . "
Lua tables are neither values nor variables; they *objects* . If you are familiar with arrays in Java or Scheme, then you should you know what I mean. You can think of a table as dynamically allocated object; your program only works with a link (pointer) to it. Lua never uses covert copying or creating new tables. Moreover, you even no need to declare a table in Lua; in fact there is not even a way declare a table. You create tables using a special expression, which in the simplest case looks like {} ;

```
a = {}
- create a table and remember the link to it in 'a'
k = "x"
a [k] = 10
- new record with key "x" and value 10
a [20] = "great" –- new record with key 20 and value "great"
print (a ["x"]) -> 10
k = 20
print (a [k]) -> "great"
a ["x"] = a ["x"] + 1 –- increase the record "x"
print (a ["x"]) -> 11
```

The table is always anonymous. There is no permanent connection between the variable that contains the table and the table itself:

```
a ["x"] = 10
b = a
- 'b' refers to the same table as 'a'
print (b ["x"]) -> 10
b ["x"] = 20
```

print (a [“x”]) -> 20
a = nil - only 'b' still refers to the table
b = nil - there are no references to the table

When there are no more references to the table left in the program, the collector garbage in Lua will eventually destroy the table and reuse it memory.
Each table can contain values with different types of dex, and the table grows as new records are added:

```
a = {}
- empty table
- create 1000 new entries
for i = 1, 1000 do a [i] = i * 2 end
print (a [9]) -> 18
a [“x”] = 10
print (a [“x”]) -> 10
print (a [“y”]) -> nil
```

Pay attention to the last line: as with the global variables, uninitialized table fields return **nil** . As with global variables, you can assign the table field is **nil** to destroy it. This is not a coincidence: Lua stores global variables in regular tables. We we'll take a closer look at this in Chapter 14. To represent records, you use the field name as an index. Lua supports this view by offering the following syntax sic sugar ": instead of a [" name "] you can write a.name . So Thus, we can rewrite the last few lines of the previous the following example in a cleaner way:

```
ax = 10 - same as a [“x”] = 10
print (ax) - same as print (a [“x”])
print (ay) - same as print (a [“y”])
```

For Lua, these two forms are completely equivalent and can be
freely used. For the reader, however, each form can
communicate a specific intent. Dot notation clearly
shows that we are using the table as a record (structure), where y
we have a certain set of given, predefined keys.
Another entry suggests that the table can be used
use any string as a key and for some reason in
this place we work with a specific key.
A common newbie mistake is that
they confuse ax and a [x] . The first form actually matches
a [“x”] , that is, referring to the table with the key “x” . In the second case
the key is the value of the variable x . Shown below

difference:

```
a = {}
x = “y”
a [x] = 10 - write 10 in the “y” field
print (a [x]) -> 10 - value of the "y" field
print (ax) -> nil - value of field “x” (undefined)
print (ay) -> 10 - value of the field "y"
```

To represent a traditional array or list, simply use
use a table with integer keys. There is no way, no
the need to declare size; you just initialize those elements
the cops you need:

```
- read 10 lines, memorizing them in the table
a = {}
for i = 1, 10 do
a [i] = io.read ()
end
```

Since you can index the table by any value-
you can start indices in an array with any number that
you like. However, it is customary in Lua to start arrays with one.
(not from scratch as in C) and some Lua tools stick to this-
th agreement.

Usually, when you are working with a list, you need to know its length.
It can be constant or it can be written somewhere. Usually
we write the length of the list in a field with a non-numeric key; by history
For logical reasons, some programs use for these purposes
Leave the field “n” .

Often, however, the length is not explicitly specified. Since anyone who does not
the value of **nil** corresponds to the initialized field , then we can use
use this value to determine the end of the list. For instance,
if you've read ten lines into a list, it's easy to remember that its
the length is 10, since its keys are the numbers 1, 2,…, 10.

This approach only works with lists that have no *holes* , which
they contain the value **nil** . We call such lists of *sequences*
sequences .

For sequences, Lua offers the '#' length operator. is he
returns the last index or length of a sequence. On-
example, you can print the lines read in the previous
example using the following code:

```
- print the lines
for i = 1, #a do
print (a [i])
end
```

Since we can index the table with values of any
type, then when indexing a table, the same subtleties arise as
and when checking for equality. Although we can index the table
and using the integer 0 , and using the string “0” , these two mean
The values are different and correspond to different elements of the table. Analogically, the lines “+1” , “01” and “1” also correspond to different elements
table cops. When you are unsure about the type of your indexes,
use explicit casting:

```
i = 10; j = “10”; k = "+10"
a = {}
a [i] = “one value”
a [j] = “another value”
a [k] = “yet another value”
print (a [i])
-> one value
print (a [j])
-> another value
print (a [k])
-> yet another value
print (a [tonumber (j)]) -> one value
print (a [tonumber (k)]) -> one value
```

If you do not pay attention to these subtleties, then it is easy to add to the program
hard-to-find errors.

2.6. Functions

Functions are first class values in Lua: programs
can write functions to variables, pass functions as
arguments to other functions and return functions as a result.
This capability lends tremendous flexibility to the language; software
ma can override the function to add new functionality
nality, or simply remove the function to create a safe
environment for executing a piece of untrusted code (for example,
code received over the network). Moreover, Lua provides a good
support for functional programming, including nested
functions with the appropriate lexical environment; simple
then wait until chapter 6. Finally, first class functions play
an important role in the object-oriented features of Lua as
we will see in chapter 16.
Lua can call functions written in Lua and functions that

written in C. Usually we use functions written in C.
C, in order to get high performance and access to the
features not available directly from Lua, such as accessing
operating system tools. All standard libraries in Lua
are written in C. They include functions for working with strings
mi, working with tables, input / output, access to basic
operating system, math functions and debugging.
We will discuss Lua functions in Chapter 5 and C functions in Chapter 27.

2.7. userdata and threads

The userdata type allows you to store arbitrary C language data in
Lua variables. This type has no built-in operations, except
assignment and equality testing. Values of this type
are used to represent new types created by applications
ny or a library written in C; for example, the standard library
the I / O library uses them to represent open
files. We will discuss this type in more detail later when we move on
to the C API.
The *thread* type will be covered in Chapter 9, where we will look at
coroutines.

Exercises

Exercise 2.1. What is the meaning of an expression
`type (nil) == nil` ? (You can use Lua to check
your answer.) Can you explain the result?
Exercise 2.2. Which of the following is acceptable
by numbers? What are their meanings?
```
.0e12 .e12 0.0e 0x12 0xABFG 0xA FFFF 0xFFFFFFFF
0x 0x1P10 0.1e1 0x0.1p1
```
Exercise 2.3. 12.7 is equal to 127/10, where all numbers
are decimal. Can you imagine it as
the value of a binary fraction? And the number 5.5?
Exercise 2.4. How would you write the following XML snippet in
a Lua string?
```
<! [CDATA [
Hello world
]]>
```
Use at least two different methods.

Exercise 2.5. Let's say you need to write a long after-
a sequence of arbitrary bytes as a string constant in
Lua. How do you do it? Pay attention to readability,
maximum line length and performance.
Exercise 2.6. Consider the following code:
```
a = {}; aa = a
```
What will be the value of `aaaa` ? Any `a` in this sequence
is somehow different from the rest?
Now add the following line to the previous code:
```
aaaa = 3
```
What will be the value of `aaaa now` ?

CHAPTER 3
Expressions

Expressions represent values. Expressions in Lua include
numeric constants and string literals, variables, unary and
binary operations and function calls. Expressions also include-
contain non-standard function definitions and constructors for
tables.

3.1. Arithmetic operators

Lua supports the standard arithmetic operators: binary-
nye `'+'` (addition), `'-'` (subtraction), `'*'` (multiplication), `'/'` (div-
number), `'^'` (exponentiation), '%' (remainder of division), and unary
'-' (change sign). All of them work with floating-point numbers.
point. For example, `x ^ 0.5` calculates the square root of `x` ,
and `x ^ (- 1/3)` computes the inverse of the cube root of `x` .
The following rule defines the modulus operator:

```
a% b == a - math.floor (a / b) * b
```
For integer operands, it has a standard value, and re-
The result has the same sign as the second operand. For real
operands, it has some additional features. On-
for example, `x% 1` gives the fractional part of x, and `xx% 1` gives the integer part.
Ana-
logical `xx% 0.01` gives `x` with exactly two decimal places after
comma:
```
x = math.pi
print (x - x% 0.01) -> 3.14
```
As another example of using the remainder operator
division, consider the following example: let's say you want to know
will the vehicle after turning at a given angle
move in the opposite direction. If the angle is given in degrees, then
you can use the following formula:
```
local tolerance = 10
function isturnback (angle)
angle = angle% 360
return (math.abs (angle - 180) <tolerance)
end
```
This definition works even for negative angles:
```
print (isturnback (-180)) -> true
```
If you want to work in radians instead of degrees, we just
let's change the constants in the functions:
```
local tolerance = 0.17
function isturnback (angle)
angle = angle% (2 * math.pi)
return (math.abs (angle - math.pi) <tolerance)
end
```
All we need is an `angle% (2 * math.pi)` operation to convert
any angle to the interval [0, 2π).

3.2. Comparison Operators

Lua provides the following comparison operators:
```
<> <=> = == ~ =
```
All of these operators are always boolean.
The == operator tests for equality; operator ~ = is negation
equality. We can use both of these operators to any two
values. If the values are of different types, then Lua assumes that
they are not equal. Otherwise Lua compares them accordingly

their type. The **nil** value is equal only to itself.
Lua compares tables and objects of type userdata by reference, i.e.
two such values are considered equal only if they are
the same object. For example, after doing the following
code:

```
a = {}; ax = 1; ay = 0
b = {}; bx = 1; by = 0
c = a
```

we get a == c , but a ~ = b .
We can only apply order operators to a pair of numbers, or
a couple of lines. Lua compares strings alphabetically, following the convention
The locale set for Lua. For example, for the Portuguese locale
Latin-1 we get “acai” <”açaí” <”acorde” . Values of types from-
personal from strings and numbers, can only be compared for equality
(and inequality).
When comparing values of different types, you need to be careful
nym: remember that “0” is different from 0 . Moreover, 2 <15 is obvious
true, but “2” <“15” is false. In case you are trying to compare
a string and a number, for example 2 <“15” , an error occurs.

3.3. Logical operators

The logical operators are **and** , **or** and **not** . As well as managing
constructs, logical operators treat **false** and **nil** as false
and all others as true values. **And** return operator
gives its first operand if it is false, otherwise it returns its
second operand. The **or** operator returns its first operand if
it is not false; otherwise it returns its second operand:

```
print (4 and 5)
-> 5
print (nil and 13)
-> nil
print (false and 13)
-> false
print (4 or 5)
-> 4
print (false or 5)
-> 5
```

Both operators (**and** and **or**) use shorthand evaluation, then
there they calculate their second operand only when it is necessary
dimo. This ensures that expressions like (type (v) == ”table” and

v.tag == "h1") will not cause errors in their computation: Lua will not try to compute v.tag when v is not a table.
A useful construct in Lua is x = x or v , which is equivalent to to the following code:

```
if not x then x = v end
```

That is, the x value is set equal to the default v if x is undefined (assuming **x is** not **false**).
Another useful construct is (a and b) or c, or simply the a and b or c , as the operator **and** a higher pri- preference than **or** . It is equivalent to the expression a? B: c in C, when provided that b is not false. For example, we can choose the maximum from two numbers x and y using the following operator:

```
max = (x> y) and x or y
```

When x> y , then the first expression in the **and** operator is true, to this it returns its second value (x), which is always tiny (since it's a number), and then the **or** operator returns its first operand, x . If the expression x> y is false, then the result of the opera- the torus **and is** also false, and so the **or** operator returns its second operand, y .
The not operator always returns a boolean value:

```
print (not nil)
-> true
print (not false)
-> true
print (not 0)
-> false
print (not not 1)
-> true
print (not not nil)
-> false
```

3.4. Concatenation

Lua refers to the concatenation operator as .. (dots). If opera- rand is a number, Lua will convert it to a string. (Some languages- ki use the '+' operator for concatenation , but Lua 3 + 5 differs from 3..5 .)

```
print (“Hello“ .. “World”)
-> Hello World
print (0 .. 1)
-> 01
print (000 .. 01)
```

-> 01
Remember that strings in Lua are immutable values.
The concatenation operator always creates a new string without changing its their operands:

```
a = “Hello”
print (a .. “World”)
-> Hello World
print (a)
-> Hello
```

3.5. Length operator

The length operator works with strings and tables. With lines he gives the number of bytes per line. With tables, it returns the length of the *pos-research* presented in the table.
There are several common idioms associated with the length operator to work with sequences.

```
print (a [#a]) - prints the last element of the sequence 'a'
a [#a] = nil - removes the last element
a [#a + 1] = v - adds 'v' to the end of the list
```

As we saw in the previous chapter, the length operator is not let's say for lists with holes (nil). It only works for software sequences, which we defined as lists without holes.
More precisely, a *sequence* is a table where the keys are give a sequence 1,…, *n* for some *n* . (Remember, that any **nil** key is not actually in the table.)
In particular, a table without numeric keys is a sequential length zero.
Over the years, there have been many proposals to expand the meaning of opera-lengths to lists with holes, but this is easier said than done.
The problem is that since a list is a table, the concept "Length" is somewhat vague. For example, consider a list produced by the following code snippet:

```
a = {}
a [1] = 1
a [2] = nil - does nothing, since a [2] is already nil
a [3] = 1
a [4] = 1
```

It is easy to say that the length of this list is four and it has a hole in index 2. However, what about the following example?

```
a = {}
```

```
a [1] = 1
a [10000] = 1
```
Should we treat this `a` as a list with 10,000 elements
tami where 9998 elements are **nil** ? Now let the program do
following:
```
a [10000] = nil
```
What happened to the length of the list? Should it be 9999,
since the program removed the last item? Or maybe
10,000, because the program just changed the value of the last element
cop to **nil** ? Or should the length be 1?
Another common suggestion is to do so
so that the `#` operator returns the number of elements in the table. This semantic
the teak is clear and well defined, but of no benefit.
Let's consider all the previous examples and imagine how much
A similar operator would be useful for algorithms that work
with lists or arrays.
Even more problematic are the **nil** values at the end of the list. Ka-
What should be the length of the following list?
```
a = {10, 20, 30, nil, nil}
```
Recall that for Lua, a **nil** field is no different from
accompanying field. Thus, the previous table is indistinguishable
ma from `{10,20,30}` ; its length is 3, not 5.
You might think that **nil** at the end of the list is a special case.
However, many lists are built by adding elements, one
after another. Any list with holes built like this
thus, simply obtained by adding **nil** to its end.
Many of the lists we use in our programs are
are sequences (for example, a line in a file cannot
be **nil**), and so most of the time the length operator is not
passive for use. If you really have to work
with lists with holes, then you'd better clearly remember the length somewhere
list.

3.6. Operator Priorities

The operator precedence in Lua is given in the table below, from the oldest
to the lowest:

```
^
not # - (unary)
```

* /%
+ -
..
<> <=> = ~ = ==
and
or

All binary operators are left associative, except '^' (exponentiation) and '..' (concatenation), which are associated tive to the right. Therefore the following expressions on the left are equivalent expressions on the right:

```
a + i <b / 2 + 1
<-->
(a + i) <((b / 2) +1)
5 + x ^ 2 * 8
<-->
5 + ((x ^ 2) * 8)
a <y and y <= z <-->
(a <y) and (y <= z)
-x ^ 2
<-->
- (x ^ 2)
x ^ y ^ z
<-->
x ^ (y ^ z)
```

When in doubt, always use parentheses. It's easier than look in the manual, and most likely later when you read this code, you will have doubts again.

3.7. Table constructors

Constructors are expressions that create and initialize tables. They are the hallmark of Lua and one of its most useful and versatile mechanisms.

The simplest constructor is an empty constructor, `{}` , which creates an empty table; we've seen this before. Constructors also initialize lists. For example, the operator

```
days = {“Sunday”, “Monday”, “Tuesday”, “Wednesday”,
“Thursday”, “Friday”, “Saturday”}
```

will initialize `days [1]` to “Sunday” (the first element constructor has index 1, not 0), `days [2]` is “Monday” etc.:

```
print (days [4]) -> Wednesday
```

Lua also offers special syntax for initialization tables by fields, as in the following example:

```
a = {x = 10, y = 20}
```

This line is equivalent to the following commands:

```
a = {}; ax = 10; ay = 20
```

The original expression is simpler and faster because Lua immediately creates
It has a table with the correct size.
Regardless of which constructor we used
to create a table, we can always add and remove fields
from her:

```
w = {x = 0, y = 0, label = ”console”}
x = {math.sin (0), math.sin (1), math.sin (2)}
w [1] = “another field” - add key 1 to table 'w'
xf = w
- add key "f" to table 'x'
print (w [“x”])
-> 0
print (w [1])
-> another field
print (xf [1])
-> another field
wx = nil
- remove field “x”
```

However, creating the table right away with the correct boo-
more efficiently and clearly.
We can mix these two initialization styles (list and by
fields) in the same constructor:

```
polyline = {color = ”blue”,
thickness = 2,
npoints = 4,
{x = 0, y = 0},
- polyline [1]
{x = -10, y = 0},
- polyline [2]
{x = -10, y = 1},
- polyline [3]
{x = 0, y = 1}
- polyline [4]
}
```

The example above also shows how contributions can be-
nest constructors one into another to represent more complex
data structures. Each of the polyline [i] elements is a table,
representing a record:

```
print (polyline [2] .x)
-> -10
print (polyline [4] .y)
-> 1
```

These two forms of constructor have their limitations. For instance,

you cannot initialize fields with negative indices
or with indices that are not identifiers. For such
goals have a different, more general format. In this format, we clearly write
we use the index as an expression between square brackets:

```
opnames = {["+"] = "add", ["-"] = "sub",
["*"] = "Mul", ["/"] = "div"}
i = 20; s = "-"
a = {[i + 0] = s, [i + 1] = s..s, [i + 2] = s..s..s}
print (opnames [s])
-> sub
print (a [22])
-> ---
```

This syntax is more awkward, but also more general: considered
previously constructor forms are special cases of this
more general syntax. The constructor `{x = 0, y = 0}` is equivalent to
`{["X"] = 0, ["y"] = 0}` , and the constructor `{"r", "g", "b"}` is equivalent to `{
[1] = "r", [2] = "g", [3] = "b"}` .
You can always put a comma after the last entry in
constructor. These commas are optional:

```
a = {[1] = "red", [2] = "green", [3] = "blue",}
```

This frees programs that generate Lua constructors from
the need to process the last element in a special way.
Finally, you can always use dot c in the constructor
comma instead of comma. I usually use semicolons to de-
division of various sections in the constructor, for example, the division of the hour
from the part formatted as a list:

```
{x = 10, y = 45; "One", "two", "three"}
```

Exercises

Exercise 3.1. What will the following program print?

```
for i = -10, 10 do
print (i, i% 3)
end
```

Exercise 3.2. What is the result of the expression `2 ^ 3 ^ 4` ?
What about `2 ^ -3 ^ 4` ?
Exercise 3.3. We can represent the polynomial
$a_n x_n + a_{n-1} x_{n-1} + \ldots + a_1 x_1 + a_0$
in Lua as a list of its coefficients $\{ a_0 , a_1 , \ldots, a_n \}$.
Write a function that receives a polynomial (represent

table) and x returns a polynomial value
ma in x .
Exercise 3.4. Can you write a function from the previous
present exercises so as to use *n* additions and *n* multiplied
scaling (and not using exponentiation)?
Exercise 3.5. How can you check if the value is
boolean without resorting to the type function ?
Exercise 3.6. Consider the following expression:
(x and y and (not z)) or ((not y) and x)
Are parentheses needed in this expression? How would you go
advised to use them in this expression?
Exercise 3.7. What will the following code snippet print?
Explain.

```
sunday = “monday”; monday = "sunday"
t = {sunday = “monday”, [sunday] = monday}
print (t.sunday, t [sunday], t [t.sunday])
```

Exercise 3.8. Suppose you want to create a table that-
paradise with each escape sequence (escape se-
quence) for strings binds its value. How would you write
a constructor for such a table?

CHAPTER 4

Operators

Lua supports an almost traditional set of operators like
to a set used in C or Pascal. Traditional operators
include assignment, control constructs and calls
procedures. Lua also supports the less common opera-
tori such as multiple assignment and locale definition
variables.

4.1. Assignment operators

Assignment is the basic means of changing the values of variables
noah and table fields:

```
a = “hello” .. “world”
tn = tn + 1
```

Lua allows *multiple assignments* , which
swarm assigns a list of values to a list of variables in one step.

For example, in the operator

```
a, b = 10, 2 * x
```

variable a gets the value `10` , and variable `b` gets the value `2 * x` .
In multiple assignment, Lua first evaluates all values.
and only then performs the assignments. Therefore, we can use
use multiple assignments to change
in places two values, as in the following examples:

```
x, y = y, x
- swap 'x' and 'y'
a [i], a [j] = a [j], a [i] - swap 'a [i]' and 'a [j]'
```

Lua always converts the number of values to the number of variables:
when the number of values is less than the number of variables, then
Lua pads the list of values with the appropriate number of **nils** ,
and when the number of values is greater, then the extra values are simply
thrown:

```
a, b, c = 0, 1
print (a, b, c)
-> 0 1 nil
a, b = a + 1, b + 1, b + 2 - the value b + 2 is discarded
print (a, b)
-> 1 2
a, b, c = 0
print (a, b, c)
-> 0 nil nil
```

The last assignment in the example above shows the distribution
an error. In order to initialize the list, re-
variables, you must provide a value for each variable:

```
a, b, c = 0, 0, 0
print (a, b, c)
-> 0 0 0
```

In fact, most of the previous examples are somewhat
then artificial. I rarely use multiple assignments
just to connect several unrelated
bout of assignments in one line. In particular, the multiple assignment
an assignment is no faster than a set of corresponding single assignments.
vanii. However, often we really need a plural
assignment. We have already seen an example that changes two variables
values. More frequent use is to obtain
several values returned by the function at once. How do we discuss
dim in section 5.1, a function can return several values at once.
In such cases, multiple assignment is usually used -
to get all these values. For example, in the assignment

a, b = f () call f gives two values: the first one is written to a ,
and the second is at b .

4.2. Local variables and blocks

In addition to global variables, Lua also supports local
variable variables. We create local variables with
operator **local** :

```
j = 10
- global variable
local i = 1 - local variable
```

Unlike global variables, the scope of local
the variable is limited to the block where it was declared. *Block* is
control structure body, function body and code block (file
or the line where the variable was declared):

```
x = 10
local i = 1 - local in the block
while i <= x do
local x = i * 2 - local inside the while block
print (x) -> 2, 4, 6, 8, ...
i = i + 1
end
if i> 20 then
local x
- local inside “then”
x = 20
print (x + 2) - (will print 22 if the condition is met)
else
print (x) -> 10 (global)
end
print (x)
-> 10 (global)
```

Please note that this example will not work like
expected if you enter it interactively. In the inter-
active mode, each line is an independent block (for
except for the case when the string is not a complete const
manual). Once you enter the second line of the example (local
i = 1), Lua will execute it and start a new block of code (next line
ka). By that time, the scope of local variable i is already
will end. To solve this problem, we can explicitly conclude
this whole block between the **do** - **end keywords** . When you enter
do , the block will end only when you enter the corresponding
end , so Lua won't try to execute each line as
separate block.

These **do** -blocks prove useful when we need BO
More precise control over the scope of local variables:

```
do
local a2 = 2 * a
local d = (b ^ 2 - 4 * a * c) ^ (1/2)
x1 = (-b + d) / a2
x2 = (-b - d) / a2
end –- scope 'a2' and 'd' ends here
print (x1, x2)
```

It is good style to use local variables
wherever possible. Local variables help
you avoid clogging the global environment with unnecessary names
mi. Moreover, accessing a local variable is faster than accessing
to the global. Finally, the local variable ceases to exist.
as soon as its scope ends, allowing the collection
the garbage bin to free the memory occupied by its value.
Lua treats local variable declarations simply as
operators. Therefore, you can insert a description of the local re-
wherever you can insert a statement. Scope
of the described variables begins immediately after the description and ends
end of the block. Each description may include an assignment
initial value, which acts the same as the operator for
piles: extra values are discarded, extra variables
get **nil** . If there is no assignment in the variable description
nil , the corresponding variable is set to **nil** :

```
local a, b = 1, 10
if a <b then
print (a) -> 1
local a - implies '= nil'
print (a) -> nil
end
- ends the block started by 'then'
print (a, b) -> 1 10
```

A common idiom in Lua is the following:

```
local foo = foo
```

This code creates a local variable foo and initializes it
the value of the global variable foo . (Local variable foo
becomes visible only after this announcement.) This idiom
useful when a block needs to store a value
the original variable, if it changes somewhere later in the code;
it also speeds up access to this variable.
Since many languages are forced to declare all local

variables at the beginning of a block (or procedure), some people think that it is bad practice to declare variables in the middle of a block. In fact, the opposite is true: declaring a variable only when you really need it, you rarely need to declare it without initial value (and therefore you are unlikely to forget its initial lyse). Moreover, you reduce the scope of change - noah, which makes the code easier to read.

4.3. Control constructs

Lua provides a small and fairly traditional set of control constructs using **if** for conditional execution and **while** , **repeat** and **for** to iterate. All control structures have an explicit ending: **end** terminates **if** , **for** and **while** , while how **until** ends **repeat** .

The conditional execution of the control structure can give any value. Remember that Lua considers all values other than nye from **false** and **nil** , as true. (In particular, Lua considers zero and an empty string as true values.)

if then else

The **if statement** checks the condition and execute its *then-part* or its *else-part* respectively. The **else** part is optional.

```
if a <0 then a = 0 end
if a <b then return a else return b end
if line> MAXLINES then
showpage ()
line = 0
end
```

For writing nested **if statements,** you can use **elseif** . This is similar to **else** followed by an **if** , but it does not no need for many **end** :

```
if op == “+” then
r = a + b
elseif op == “-” then
r = a - b
elseif op == “*” then
r = a * b
elseif op == “/” then
r = a / b
else
error (“invalid operation”)
```

```
end
```
Since there is no *switch statement* in Lua , such constructions can be are freely common.

while

As the name suggests, this operator repeats its body while the condition is true. As usual, Lua checks the condition first; if a it is false, then the cycle ends; otherwise Lua does the body of the loop and repeats the given process.

```
local i = 1
while a [i] do
print (a [i])
i = i + 1
end
```

repeat

As the name suggests, the **repeat** - **until** statement repeats its body until the condition becomes true. Checking the condition is added after the loop body is executed, so the loop body will be executed at least once.

```
- print the first non-empty line
repeat
line = io.read ()
until line ~ = “”
print (line)
```

Unlike many languages, in Lua the scope of local variables include the loop condition:

```
local sqr = x / 2
repeat
sqr = (sqr + x / sqr) / 2
local error = math.abs (sqr ^ 2 - x)
until error <x / 10000 - local variable 'error' is visible here
```

Numeric for statement

The **for** statement **comes** in two flavors - numeric **for** and generic **for** .

The numeric **for** statement looks like this:

```
for var = exp1, exp2, exp3 do
```

<something>
end
This loop will execute *something* for every var value from
exp1 to exp2 , using exp3 as a *step* to increase var . This is a
your expression (exp3) is optional; when absent, Lua
uses 1 as a step. As typical examples of such
cycles can be considered
for i = 1, f (x) do print (i) end
for i = 10, 1, -1 do print (i) end
If you want to get a loop without an upper limit, then you can
use constant math.huge :
for i = 1, math.huge do
if (0.3 * i ^ 3 - 20 * i ^ 2 - 500> = 0) then
print (i)
break
end
end
The **for** loop has some subtleties that you know best,
to use it well. First, all three expressions are
are counted only once, before the start of the cycle. For example, in our
In the first example, Lua will execute f (x) just once. Secondly,
the control variable is a local variable, automatic
cally declared operator **for** , and it is only within the visible
cycle. It is a common mistake to think that this variable is all
still exists after the end of the loop:
for i = 1, 10 do print (i) end
max = i - possibly wrong! Here 'i' is global
If you need the value of the control variable after the loop
(usually when you exit the loop ahead of time), then you should
store its value in another variable:
- find a value in the list
local found = nil
for i = 1, #a do
if a [i] <0 then
found = i - save value of 'i'
break
end
end
print (found)
Third, you should never change the value of the control
variable: the effect of such changes is unpredictable. If you
want to end the **for** loop to terminate it normally, use
Call **break** (as we did in the previous example).

General for statement

The generic **for** statement iterates over all the values returned by ite-
ruling function:

```
- print all values in table 't'
for k, v in pairs (t) do print (k, v) end
```

This example uses pairs , a convenient iteration function
to traverse the entire table provided by the underlying library
Lua. At each step of this cycle, k gets an index, and v gets a value,
associated with this index.

Despite its seeming simplicity, the general **for** operator -
it is a very powerful language construct. With suitable iterators
you can bypass just about anything you want in an easy-to-read form-
me. The standard libraries provide multiple iterators,
allowing us to iterate over the lines of the file (io.lines), pairs from the tab-
faces (pairs), elements of sequence (ipars), words inside
strings (string.gmatch), etc.

Of course, we can write our own iterators as well. Although
using the **for** statement in its general form is easy, the task of writing
an iterator function has its own subtleties; we will cover this topic
later in chapter 7.

The general loop operator has two features in common with
numeric loop operator: loop variables are local to
loop bodies, and you should never write any
values.

Let's consider a more specific example of using the operator
for general view. Let's say you have a table with the names of non-
Delhi:

```
days = {“Sunday”, “Monday”, “Tuesday”, “Wednesday”,
“Thursday”, “Friday”, “Saturday”}
```

Now you want to translate the name of the day to its position in the week.
You can traverse the entire table looking for a given name. But,
as you'll soon find out, you rarely need to search in Lua. More efficient
an effective approach would be to build a reverse table, for example
revDays , where day names are indices and values are
mi are day numbers. This table will look like this
way:

```
revDays = {[“Sunday”] = 1, [“Monday”] = 2,
[“Tuesday”] = 3, [“Wednesday”] = 4,
```

```
["Thursday"] = 5, ["Friday"] = 6,
["Saturday"] = 7}
```
Then all you need to find the day number is
refer to this reverse table:
```
x = "Tuesday"
print (revDays [x]) -> 3
```
Of course, you don't have to explicitly set this reverse table. We can
build it automatically from the original:
```
revDays = {}
for k, v in pairs (days) do
break, return and goto
revDays [v] = k
end
```
This loop will perform an assignment for each element of `days` ,
where the variable `k` gets the key (1, 2, ...) and `v` gets the value
("Sunday", "Monday", ...).

4.4. break, return and goto

The **break** and **return statements** allow us to jump right out of the block.
ka. The **goto statement** allows us to jump to almost any
place of function.
We use the **break** statement to end the loop. This opera-
the generator interrupts the inner loop (**for** , **repeat** or **while**) containing
shying him. Also it can be used to return from a function,
so you don't have to use the **return statement** unless you
return no value.
For syntactic reasons, the **return statement** can only be
to the last statement of the block: in other words, or the last
operator, or right before **end** , **else,** or **until** . In the following case-
as the **return statement** - the last statement block **the then** .
```
local i = 1
while a [i] do
if a [i] == v then return i end
i = i + 1
end
```
This is usually the place where we use **return** , because
any other operators following it would never
filled up. Sometimes it's actually useful to write **return** in
middle of the block; for example you can debug a function and want
avoid doing it. In cases like this, you can use

put an explicit **do** block around the **return statement** :

```
function foo ()
return
- << SYNTAX ERROR
- 'return' is the last statement in the next block
do return end
- OK
<other statements>
end
```

The **goto statement** translates program execution to the appropriate label. There were long discussions about **goto** , some people even now they believe that they are harmful to programming and should us to be excluded from programming languages. However, many languages offer a similar operator, and they have a reason for it. These operators are a powerful mechanism that will used carefully, can improve the quality of your th code.

In Lua, the syntax for the **goto statement** is quite traditional: it is a **goto** reserved word followed by a label name which can be any valid identifier. Syntax for the label, however, is more complex: it consists of two colons, followed by followed by the label name followed by two more colons, for example :: name :: . This complexity is intentional, its purpose is to force programmer think twice before using **goto** .

Lua puts some restrictions on where you can jump with **goto** . First, the labels follow the usual rules of visibility, so you cannot jump directly inside the block (since the label inside the block is invisible outside of it). In- second, you cannot jump out of the function. (Note, that the first rule excludes the possibility of jumping *inside the* function tion.) Third, you cannot jump inside the area of effect of the lo- local variable.

A typical and well-used use of the **goto statement** is an emulation of some construction you learned from in another language, but which is absent in Lua, such as **continue** , many go-level **break** , **redo** , etc. The **continue** statement is just a transition to the label at the end of the loop, the **redo** statement jumps to the beginning of the block:

```
while some_condition do
:: redo ::
if some_other_condition then goto continue
```

```
else if yet_another_condition then goto redo
end
<some code>
:: continue ::
end
```

A useful nuance in the Lua specs is that the scope

```
while some_condition do
if some_other_condition then goto continue end
local var = something
```

of a local variable ends with the last *nepu-*
stom operator of the block where the variable is defined; tags are counted
empty operators. In order to see the usefulness of this,
look at the following code snippet:

```
break, return and goto
<some code>
:: continue ::
end
```

You might think this **goto statement** jumps straight
to the scope of `var` . However, the `continue` label finds-
after the last non-empty block operator, and therefore not in the region
These are the actions of `var` .
The **goto statement is** also useful when writing state machines.
As an example, Listing 4.1 is an example program that
which believes whether its input contains an even number of zeros. Exists
better ways of writing this program, but this approach
very useful if you want to automatically translate the ending auto-
tomato to Lua code (think about automatic code generation).
As another example, consider a simple maze game.
The maze contains several rooms, each with up to four doors:
north, south, east and west. At each step, the user enters
movement board. If there is a door in this direction, then the user
the provider enters the appropriate room; otherwise the print program
there is a warning. The goal is to get from the starting room to
the ultimate room.
This game is a typical machine where the current room is
is a state. We can implement this game using one
a block for each room and a **goto statement** to jump from one
rooms to another. Listing 4.2 shows how you can write a simple
the smallest labyrinth of four rooms.
For this simple game, you might decide that the program that controls
data when you describe rooms and movements when

using tables is a better solution. However, if in
each room has its own characteristics, then this approach provides
is quite successful.

Listing 4.1. State machine example using **goto**

```
:: s1 :: do
local c = io.read (1)
if c == '0' then goto s2
elseif c == nil then print'ok '; return
else goto s1
end
end
```

```
:: s2 :: do
local c = io.read (1)
if c == '0' then goto s1
elseif c == nil then print'not ok '; return
else goto s2
end
end
goto s1
```
Listing 4.2. Maze game
```
goto room1 - starting room
:: room1 :: do
local move = io.read ()
if move == “south” then goto room3
elseif move == “east” then goto room2
else
print (“illegal move”)
goto room1 - stay in the same room
end
end
:: room2 :: do
local move = io.read ()
if move == “south” then goto room4
elseif move == “west” then goto room1
else
print (“illegal move”)
goto room2
end
end
:: room3 :: do
local move = io.read ()
if move == “north” then goto room1
elseif move == “east” then goto room4
else
print (“illegal move”)
goto room3
end
end
:: room4 :: do
print (“Congratulations, you won!”)
end
```

Exercises

Exercise 4.1. Most languages with C-like syntax
catfish doesn't offer an **elseif** construct . Why is this construct
Is it more needed in Lua than in other languages?

Exercise 4.2. Write four different ways to implement

to create an unconditional loop in Lua. Which one do you prefer curls?

Exercise 4.3. Many people think that **repeat - until is** used rarely and therefore should not be present in minimalist languages like Lua. What do you think about it?

Exercise 4.4. Rewrite the state machine in Listing 4.2. without using **goto** .

Exercise 4.5. Can you explain why Lua has Is there a restriction that you can't jump out of the function? (Hint: How would you implement this feature?)

Exercise 4.6. Assuming goto can jump out of functions, explain that the program in Listing 4.3 should make. (Try to reason about the label using the same rules that are used to describe the the scope of local variables.)

Listing 4.3. Strange (and incorrect) **goto** usage

```
function getlabel ()
return function () goto L1 end
:: L1 ::
return 0
end
function f (n)
if n == 0 then return getlabel ()
else
local res = f (n - 1)
print (n)
return res
end
end
x = f (10)
x ()
```

CHAPTER 5
Functions

Functions are the main mechanism for abstraction of operators and expressions in Lua. Functions can perform a specific task (in other languages this is often called a *procedure* or *subroutine*) or calculate and return values. In the first case, we use you function call as operator; in the second case we use it as expression:

```
print (8 * 9, 9/8)
a = math.sin (3) + math.cos (10)
print (os.date ())
```

In both cases, the argument list is enclosed in parentheses, indicating a call; if the function has no arguments, then we are all equal but must write `()` to indicate a function call. Existence

There is a special exception to this rule: if a function has all one argument and this argument, either a literal (character string) or table constructor, the parentheses are optional:

```
print "Hello World"
<--> print (“Hello World”)
dofile 'a.lua'
<--> dofile ('a.lua')
print [[a multi-line
<--> print ([[a multi-line
message]] message]])
f {x = 10, y = 20}
<--> f ({x = 10, y = 20})
type {}
<--> type ({})
```

Lua also offers special syntax for object-directed calls, colon operator. An expression like `o: foo (x)` is just a way to write `o.foo (o, x)` , that is, to call `o.foo` by adding `o` as an optional argument. In Chapter 16, we discuss we make similar calls (and object-oriented programming in more detail.

A Lua program can use functions written as in Lua and C (or any other language used by the application niy). For example, all functions from the Lua standard library are written sana in C. However, when calling a function, there is no difference between functions written in Lua and functions written in C.

As we saw in other examples, the function definition follows traditional syntax like below:

```
- add the elements of the sequence 'a'
function add (a)
local sum = 0
for i = 1, #a do
sum = sum + a [i]
end
return sum
end
```

In this syntax, the function definition contains a *name* (in the example add), a list of *parameters*, and a *body* that is a list of operators

moat.
Parameters work as local variables, initials
values of arguments passed when calling the function
tion. You can call a function with a number of arguments, different
from its parameter list. Lua will cast the number of arguments
to the number of parameters in the same way as it is done in the plural
piling: unnecessary arguments are discarded instead of missing ones
nil is added . For example, consider the following function:

```
function f (a, b) print (a, b) end
```

She has this behavior:

```
f (3) -> 3 nil
f (3, 4) -> 3 4
f (3, 4, 5) -> 3 4 (5 is discarded)
```

Although this behavior can lead to errors (easy
at runtime), it is also useful, especially
for default arguments. For example, consider the following
a function that increments the global counter:

```
function incCount (n)
n = n or 1
count = count + n
end
```

This function has one default parameter; if we call
we put it `incCount ()` without arguments, then it will increment `count` by one.
When you call `incCount ()` , Lua first initializes `n` values
by **nil** ; the `or` operator returns its second argument, and as a result
Lua sets the variable `n` to 1.

5.1. Multiple results

A little common, but nevertheless very convenient feature
The thing about Lua is that a function can return multiple values.
Some of the predefined functions in Lua return multiple
values. As an example, take the `string.find` function ,
which looks for a pattern in a string. This function returns two indices
sa when it finds a pattern: the index of the start of the pattern in the string and the index
end of the pattern. Multiple assignment allows the program
get both results:

```
s, e = string.find (“hello Lua users”, “Lua”)
print (s, e)
```

-> 7 9
(Note that the index of the first character of the string is 1.)
Functions we write ourselves can also return immediately
multiple values by simply listing them after the word **return** . On-
example, a function that searches for the maximum element in a sequence
value, can return both the maximum element itself and its
index:

```
function maximum (a)
local mi = 1
- index of the maximum element
local m = a [mi]
- maximum value
for i = 1, #a do
if a [i]> m then
mi = i; m = a [i]
end
end
return m, mi
end
print (maximum ({8,10,23,12,5})) -> 23 3
```

Lua always lists the number of values returned by a function.
her, to the circumstances of her call. When we call the function like
operator, then Lua discards all return values. When we
we use a function call in an expression, then Lua only stores
first value. We get all the values only when the function call
is the last (or only) expression in the list
expressions. In Lua, these lists appear in four constructs:
multiple assignment, function call arguments, const
table handler and **return statement** . To illustrate all these
cases, we consider the following function definitions:

```
Multiple results
function foo0 () end
- returns nothing
function foo1 () return “a” end - returns 1 value
function foo2 () return “a”, “b” end - returns 2 values
```

In multiple assignment, function call as last
(or a single) expression uses as many results as
how much is needed to match the list of variables:

```
x, y = foo2 ()
- x = ”a”, y = ”b”
x = foo2 ()
- x = ”a”, “b” is discarded
x, y, z = 10, foo2 ()
```

- x = 10, y = ”a”, z = ”b”

If the function returns no value or returns, but not so
as much as required, then use
is **nil** :

```
x, y = foo0 ()
- x = nil, y = nil
x, y = foo1 ()
- x = ”a”, y = nil
x, y, z = foo2 ()
- x = ”a”, y = ”b”, z = nil
```

A function call that is not the last item in
list, gives exactly one value:

```
x, y = foo2 (), 20
- x = ”a”, y = 20
x, y = foo0 (), 20, 30
- x = nil, y = 20, 30 is discarded
```

When the function call is the last (or only) one
argument of another call, then all results of the first call go
as arguments to the input of the second call. We have already seen examples of this
constructs with the print function . Since the print function can
receive a variable number of arguments, the operator print (g ()) prints
melts all the values returned by g .

```
print (foo0 ())
->
print (foo1 ())
-> a
print (foo2 ())
-> ab
print (foo2 (), 1)
-> a 1
print (foo2 () .. “x”)
-> ax (see below)
```

When a call to foo2 is inside an expression, Lua
converts the number of returned values to one; therefore in the last
on this string, concatenation only uses “a” .
If we write f (g (x)) and f has a fixed number of arguments,
then Lua converts the number of values returned to the number of arguments f ,
as we saw earlier.
The table constructor also uses all values returned by
by function, without any changes:

```
t = {foo0 ()}
- t = {} (empty table)
t = {foo1 ()}
```

- t = {“a”}
t = {foo2 ()}
- t = {“a”, “b”}

As always, this behavior occurs only if the call is
the last expression in the list; calls anywhere else
give exactly one value:

t = {foo0 (), foo2 (), 4} - t [1] = nil, t [2] = “a”, t [3] = 4

Finally, the return f () statement returns all values that
returned f :

```
function foo (i)
if i == 0 then return foo0 ()
elseif i == 1 then return foo1 ()
elseif i == 2 then return foo2 ()
end
end
print (foo (1))
-> a
print (foo (2))
-> ab
print (foo (0))
- (no values)
print (foo (3))
- (no values)
```

You can "force" the call to return only one value by
enclosing it in an extra pair of parentheses:

```
print ((foo0 ()))
-> nil
print ((foo1 ()))
-> a
print ((foo2 ()))
-> a
```

Be careful: the **return statement** does not require parentheses around
return value. So an expression like return (f (x))
always returns exactly one value, regardless of whether
how many values the function f returns . Sometimes that's exactly what
you need; sometimes not.

A special function that returns multiple values is
I wish to set up table.unpack . It takes an array as input and returns everything
elements of this array, starting at 1:

print (table.unpack {10,20,30}) -> 10 20 30
a, b = table.unpack {10,20,30} - a = 10, b = 20, 30 is discarded

An important use of unpack is a generic mechanism
function call. The generalized mechanism allows you to call any
buoy function with any arguments dynamically. In ANSI C, for example

measures, there is no way to build a generalized call. You can
declare a function that takes a variable number of arguments
(with stdarg.h) and you can call various functions,
using function pointers. However, you cannot call a function
with a variable number of arguments: every time you write
those in C, you have a fixed number of arguments, and each argument
is of a fixed type. In Lua, if you want to call a function
f with a variable number of arguments from array a , you can simply
write the following:

```
f (table.unpack (a))
```

The unpack function call returns all values from a that become
are given by the arguments to f . For example consider the following
call:

```
print (string.find (“hello”, “ll”))
```

You can dynamically build an equivalent call when
the power of the following code:

```
f = string.find
a = {“hello”, “ll”}
print (f (table.unpack (a)))
```

Usually unpack uses the length operator to find out if
how many items should be returned, so it only works with
sequences. If necessary, then it can be explicitly limited:

```
print (table.unpack ({“Sun”, “Mon”, “Tue”, “Wed”}, 2, 3))
-> Mon Tue
```

Although the unpack function is written in C, we can write it in Lua,
using recursion:

```
function unpack (t, i, n)
i = i or 1
n = n or #t
if i <= n then
return t [i], unpack (t, i + 1, n)
end
end
```

The first time we call it with a single argument
volume, i writes 1, and n writes the length sequentially
sti. The function then returns t [1] along with all the results
unpack (t, 2, n) , which in turn returns t [2] and all the results
call unpack (t, 3, n) and so on, stopping after n items
Comrade

5.2. Variable functions number of arguments

A Lua function can have an arbitrary number of arguments (*variadic*). For example, we've already called the print function with one, two name and a large number of arguments. Although print is defined in C, we and in Lua, we can write functions with a variable number of arguments.
As the next example, the function below returns the sum all its arguments:

```
function add (...)
local s = 0
for i, v in ipairs {...} do
s = s + v
end
return s
end
print (add (3, 4, 10, 25, 12)) -> 54
```

Three dots (...) in the parameter list indicate that this function has a variable number of arguments. When we call such a function tion, Lua collects all of its arguments into a list; we call these assembled arguments can *additional arguments* function. Function can access their additional ones again with three dots, now as an expression. In our example- re expression {...} gives an array with all the arguments collected. The function iterates over the elements of this array in order to find their amount.
We call expression ... *variable expression arguments* (vararg expression). It behaves like a function that returns giving many values, returning all additional arguments current function. For example, the print (...) command will print everything before additional arguments to the current function. Similarly, the following the command will create two local variables with the values of the first two additional arguments (or **nil** if there are no such arguments).

```
local a, b = ...
```

In fact, we can mimic the standard mechanism for passing parameters to Lua, translating the following construction

function foo (a, b, c)
in

function foo (...)
local a, b, c = ...
For those who like Perl's parameter passing mechanism, this is a
like.
The function shown below simply returns all passed ar-
guments:
function id (...) return ... end
The following function behaves the same as foo , except
the fact that before calling her, she prints a message with all transmitted
arguments:
function foo1 (...)
print (“calling foo:”, ...)
return foo (...)
end
This is a pretty useful trick for keeping track of all calls to
this function.
Let's take a look at another useful example. Lua provides
There are separate functions for formatting text (string.format)
and for writing text (io.write). It's quite easy to combine the two
functions into one function with a variable number of arguments:
function fwrite (fmt, ...)
return io.write (string.format (fmt, ...))
end
Note the presence of the fmt parameter in front of the periods.
Functions with a variable number of arguments can have any number of
number of fixed parameters before the part with variable number
scrap parameters. Lua assigns the first values to these variables;
the rest (if any) go as additional parameters. Below we
we will show several examples of calls and corresponding parameters
moat:
Call
Options
fwrite ()
fmt = nil, no additional parameters
fwrite (“a”)
fmt = “a”, no additional parameters
fwrite (“% d% d”, 4, 5) fmt = “% d% d”, optional 4 and 5
(Note that calling fwrite () will result in an error,
since string.format requires a string as its first argument.)
The function can be used to bypass all additional parameters.

use the expression {...} to collect them all in a table
tsu, as we did in the add function definition .
In rare cases where the passed arguments may take
value of **nil** , table created using {...} will not nastoya-
sequence. For example, there is no way for
in order to find out from this table whether there were any arguments at the end of the list.
tov **nil** 's. For these cases, Lua offers the table.pack function 1 . This
the function takes an arbitrary number of arguments and returns a new
a table containing all its arguments, like {...} , but this
the table will have an additional field n containing the total number of its arguments. The following function uses table.pack to
so that its arguments include **nil** values .

```
function nonils (...)
local arg = table.pack (...)
for i = 1, arg.n do
if arg [i] == nil then return false end
end
return true
end
print (nonils (2,3, nil))
-> false
print (nonils (2,3))
-> true
print (nonils ())
-> true
print (nonils (nil))
-> false
```

Remember, however, that {...} is faster and cleaner than table.pack .

5.3. Named arguments

The mechanism for passing parameters in Lua is *positional* : when
we call the function, then the correspondence between arguments and form-
the minimum parameters is carried out according to their position. First
the argument gives the value to the first parameter, and so on. Sometimes, however,
it is useful to specify the parameter by name. To illustrate
To complete this, let's look at the os.rename function (from the library os), which renames the file. Quite often we forget
which name comes first, new or old; so we might want

override this function so that it receives two named
parameters:

```
- wrong
rename (old = ”temp.lua”, new = ”temp1.lua”)
```

Lua has no direct support for this syntax, but
we can achieve the desired effect with a small syntax
sic change. The idea is to collect everything
arguments to a table and use this table as the only one
function argument. The special syntax that Lua provides
is used to call a function, with the table constructor as the only one
argument will help us achieve this:

```
rename {old = ”temp.lua”, new = ”temp1.lua”}
```

Accordingly, we override the rename function with only
one parameter and we get the real arguments from this para-
meters:

```
function rename (arg)
return os.rename (arg.old, arg.new)
end
```

This method of passing parameters is especially useful when functions
There are many arguments and most of them are optional. For example
measures, the function that creates a new window in the GUI library can
have dozens of arguments, most of which are optional,
and it is best to pass them using names:

Listing 5.1. Function with named optional parameters

```
function Window (options)
- check mandatory options
if type (options.title) ~ = “string” then
error (“no title”)
elseif type (options.width) ~ = “number” then
error (“no width”)
elseif type (options.height) ~ = “number” then
error (“no height”)
end
- everything else is optional
_Window (options.title,
options.x or 0, - default value
options.y or 0, - default value
options.width, options.height,
options.background or “white”, - default value
options.border - default value false (nil)
)
end
w = Window {x = 0, y = 0, width = 300, height = 200,
title = “Lua”, background = ”blue”,
border = true
```

}

Exercises

Exercise 5.1. Write a function that gets produced
any number of strings and returns them concatenated together.
Exercise 5.2. Write a function that receives an array
and prints all the elements of this array. Consider the benefits
advantages and disadvantages of using table.unpack in this
function.
Exercise 5.3. Write a function that gets produced
any number of values and returns all but the first.
Exercise 5.4. Write a function that receives an array and
prints all combinations of elements in this array.
(*Hint* : you can use a recursive formula
for the number of combinations: $C(n, m) = C(n - 1; m - 1) + C(n - 1, m)$.
To get all $C(n, m)$ combinations of n elements in
group of size m, you first add the first element to the re-
result and then generate all $C(n-1, m-1)$ combinations from
the remaining elements in the remaining places. When n *is* less,
than m , there are no more combinations. When m is zero, there is
there is only one combination, and it does not use any
items.)

CHAPTER 6
More about functions

Functions in Lua are first class values with appropriate
lexical scope.
What does it mean that functions are "first class values"?
This means that in Lua, a function is a value that has the same
rights as standard values for numbers and strings. We can
save functions in variables (local and global) and in tab-
persons, we can pass functions as arguments and return
them from other functions.
What does "lexical scope" mean for functions? This is a sign

cheat that functions can access variables containing them
functions [1] . As we will see in this chapter, this is a kind of harmless property.
v gives tremendous power to the language because it allows Lua to use
many powerful tricks from the world of functional software
ming. Even if you are not at all interested in functional
programming, it is still worth learning a little about how to use
take advantage of these opportunities as they can make your
Rammu is smaller and simpler.
A somewhat confusing concept in Lua is that functions,
like other values, are anonymous; they have no names. When
we are talking about a function name like `print` , we mean re-
a belt that contains this function. As with any other
variable containing any other value, we can manipulate
manipulate these variables in many different ways. Following
the example, although somewhat contrived, shows possible examples:

```
a = {p = print}
ap ("Hello World") -> Hello World
print = math.sin - 'print' now refers to sine
ap (print (1)) -> 0.841470
sin = ap
- 'sin' now refers to the print function
sin (10, 20)
-> 10 20
```

(We'll see useful uses of this feature later.)
If functions are values, then do there exist expressions
which create functions? Yes. In particular, the standard way
create a function in Lua like for example
`function foo (x) return 2 * x end` ,
this is just an example of what we call *syntactic sugar* ;
it's just a nicer way to write the following code:

```
foo = function (x) return 2 * x end
```

Therefore, the definition of a function is actually the operator
(assignment), which creates a value of type "function" and
assigns it to a variable. We can consider the expression
`function (x)` *body* `end` as a function constructor, just like
`{}` is a table constructor. We call the result
the definition of such constructors with *an anonymous function* . Although we
are
then we assign functions to global variables, giving them something
like a name, there are times when functions remain anonymous.
Let's take a look at some examples.

The table library provides the table.sort function , which
paradise gets a table and sorts its elements. Similar function
should provide infinite sort order variations:
ascending and descending, numeric or alphabetically, as
my key, etc. Instead of trying to provide all possible options
sort provides an additional parameter which is
an ordering function : a function that takes two arguments and
determines if the first element should be sorted before the second
tied list. For example, let's say we have the following
record table:

```
network = {
{name = “grauna”, IP = “210.26.30.34”},
{name = “arraial”, IP = “210.26.30.23”},
{name = “lua”, IP = “210.26.23.12”},
{name = “derain”, IP = “210.26.23.20”},
}
```

If you want to sort the table by name field in reverse
alphabetically, then you can simply write:

```
table.sort (network, function (a, b) return (a.name> b.name) end)
```

See how convenient it was to use anonymous function
in this statement.
A function that takes another function as an argument is
is what we call *a higher-order function* . High functions
of the following order are a convenient software mechanism, and
using anonymous functions to create their functional
arguments is a great source of flexibility. However, remember-
the thread that higher-order functions are not something special,
they are simply a consequence of Lua's ability to handle functions like
values of the first class.
In order to illustrate the use of the functions of high
orders, we will write a simplified definition of
higher order function, derivative. Following informal
definition, the derivative of the function *f* at the point *x* is the value
(*f* (*x* + *d*) - *f* (*x*)) / *d* when *d* becomes infinitesimal. According to
and with this definition, we can write an approximate value
derivative as follows:

```
function derivative (f, delta)
delta = delta or 1e-4
return function (x)
return (f (x + delta) - f (x)) / delta
end
```

```
end
```
Having received the function f , the call to derivative (f) will return an approximate the value of its derivative, which is another function:
```
c = derivative (math.sin)
> print (math.cos (5.2), c (5.2))
-> 0.46851667130038
0.46856084325086
print (math.cos (10), c (10))
-> -0.83907152907645
-0.83904432662041
```
Since functions are first class values in Lua,
we can remember them not only in global variables, but also in
local variables and table fields. As we will see later, the
using functions in table fields is a key component
some of the advanced features of Lua, such as modules and objects
ect-oriented programming.

6.1. Closures

When we write a function that is enclosed within another function,
then it has full access to the local variables of the environment
its functions; we call this *lexical scope* (lexical
scoping). While this rule of visibility may seem obvious,
in fact it is not. Lexical scope together with
functions that are first class objects is
a very powerful concept in a programming language, but many
languages don't support this.
Let's start with a simple example. Suppose you have a list of names
students and a table comparing their grades to them; you want to suck
list students according to their grades, students with higher
grades should come before. You can achieve this by following-
in a way:
```
names = {“Peter”, “Paul”, “Mary”}
grades = {Mary = 10, Paul = 7, Peter = 8}
table.sort (names, function (n1, n2)
return grades [n1]> grades [n2]
-– compare grades
end)
```
Now let's say that you want to create a function to solve the given
noah tasks:
```
function sortbygrade (names, grades)
table.sort (names, function (n1, n2)
return grades [n1]> grades [n2]
-– compare grades
```

```
end)
end
```
An interesting feature in this example is that anonymous
the function passed to the `sort` function accesses the parameter
`grades` , which is local to the enclosing function
`sortbygrade` . Within this anonymous function, `grades is` not
neither a global variable nor a local variable, but the fact that we
called a *nonlocal variable* . (For historical reasons, for
notation for non-local variables in Lua is also used term
min *upvalue* .)
Why is this so interesting? Because functions are important
first class, and so they can *leave* primary education
the power of the action of its variables. Consider the following example:
```
function newCounter ()
local i = 0
return function ()
-– anonymous function
i = i + 1
return i
end
end
c1 = newCounter ()
print (c1 ()) -> 1
print (c1 ()) -> 2
```
In this code, the anonymous function refers to the non-local re-
variable `i` to account for the value. However, by the time we call
anonymous function, variable `i` will already leave its scope by
validity, since the function that created this variable (`new-`
`Counter`) has already finished. However, Lua correctly handles
this situation using the concept of a *closure* . Simply put,
a closure is a function plus whatever it needs to access non-
local variables. If we call `newCounter` again , then it
will create a new local variable `i` , so we get a new one
a closure working on this new variable:
```
c2 = newCounter ()
print (c2 ()) -> 1
print (c1 ()) -> 3
print (c2 ()) -> 2
```
Thus, `c1` and `c2` are different closures of the same
functions, and each uses its own independently instantiated
local variable `i` .
In fact, in Lua, the meaning is a closure, not a function.

tion. A function is just a prototype for a closure. However, we
we will use the term "function" to denote the closure
whenever it will not lead to confusion.
Closures prove to be a very handy tool in many
cases. As we have already seen, they turn out to be convenient as
arguments to higher-order functions such as sort . Closures
also useful for functions that build other functions like
the newCounter function in our example, or a function for finding
denia derivative; this mechanism allows Lua programs to use
use advanced techniques from the world of functional software
ming. Closures are also handy for various *callable
functions (callback). A typical example occurs when you create
various buttons in its library to create a GUI. Each
the button has its own function, which should be called when
the user clicks on this button; usually needed for different
buttons led to different actions. For example, a calculator
you need ten buttons, one for each number. You can co-
build with a similar function:

```
function digitButton (digit)
return Button {label = tostring (digit),
action = function ()
add_to_display (digit)
end
}
end
```

In this example, we will assume that Button is a function from
a library that creates new buttons; label is the label of the button
ki; action is the closure to be called when the button is
children pressed. The closure can be noticeable after a long time
after digitButton is executed, and after locale
naya variable digit went out of their field of sight, but it is not
less, the closure can still access it.
Closures are also useful in a very different case.
tea. Since functions are stored in regular variables, we can
it is easy to override functions in Lua, including even standard ones. This
possibility is one of the reasons why Lua is so flexible. Hour-
then when you override the function you still need the old one
function. For example, you want to override the sin function to
she worked in degrees instead of radians. This new function converts
expands its argument and then calls the original sin function to produce

completing work. Your code might look like this
below:

```
oldSin = math.sin
math.sin = function (x)
return oldSin (x * math.pi / 180)
end
```

The following is a slightly neater way to accomplish this
override:

```
do
local oldSin = math.sin
local k = math.pi / 180
math.sin = function (x)
return oldSin (x * k)
end
end
```

We now store the old version in a local variable;
the only way to access it is through a new function.
You can use this same approach to create secure
environments, also called *sandboxes* . Safe
environments are extremely important when executing code from untrusted sources.
sources such as the Internet. For example, to restrict files,
which the program can access, we can override
the `io.open` function using closures:

```
do
local oldOpen = io.open
local access_OK = function (filename, mode)
<check access>
end
io.open = function (filename, mode)
if access_OK (filename, mode) then
return oldOpen (filename, mode)
else
return nil, “access denied”
end
end
end
```

What makes this example especially enjoyable is that after that
overrides there is absolutely no way for the program to call
the original `open` , except through a new version with control. Nebezo-
the passable version is stored in a local variable inside the closure,
not attainable from the outside in any way. With this approach, you can build
sandboxes for Lua on Lua itself, while receiving as advantages
simplicity and flexibility. Instead of some kind of one-stop solution for

all problems Lua provides a meta mechanism, so you can
adjust your environment to your goals.

6.2. Non-global functions

The obvious consequence of the fact that functions are values
first class, is that we can save functions not
only in global variables, but also in local variables-
data and fields of the table.
We have already seen various examples of functions stored in fields.
tables: most Lua libraries use this mechanism (by
example, io.read , math.sin). To create similar functions in Lua
we just need to combine the standard syntax for functions with
syntax for tables:

```
Lib = {}
Lib.foo = function (x, y) return x + y end
Lib.goo = function (x, y) return x - y end
print (Lib.foo (2, 3), Lib.goo (2, 3)) -> 5 -1
```

Of course, we can also use constructors:

```
Lib = {
foo = function (x, y) return x + y end,
goo = function (x, y) return x - y end
}
```

Moreover, Lua also provides another syntax for
similar functions:

```
Lib = {}
function Lib.foo (x, y) return x + y end
function Lib.goo (x, y) return x - y end
```

When we store a function in a local variable, we get
we tea a *local function* , that is, a function with a limited scope
visibility. Definitions like this are especially handy for packages:
since Lua treats each block as a function, a block can
Can define local functions that are only visible from the block.
Lexical scoping ensures that other functions from
package can use these local functions:

```
local f = function ( <params> )
<body>
end
local g = function ( <params> )
<some code>
f () - 'f' is visible here
<some code>
```

```
end
```
Lua also supports the following syntactic sugar for lo-
cal functions:
```
local function f ( <params> )
<body>
end
```
When defining recursive local functions, one gets
subtlety. The naive approach doesn't work here. Consider the following
definition:
```
local fact = function (n)
if n == 0 then return 1
else return n * fact (n-1) - error
end
end
```

When Lua compiles the `fact (n-1)` call in the body of a function, then the lo-
The actual function `fact` has not yet been defined. Therefore, this definition
it will try to call the global function `fact` , not the local one.
We can solve this problem by first defining a local re-
variable and then already defining the function itself:
```
local fact
fact = function (n)
if n == 0 then return 1
else return n * fact (n-1)
end
end
```
Now `fact` inside the function refers to a local variable.
Its value at the moment the function is defined does not mean anything;
by the time the function is executed, it will already receive
correct value.
When Lua "reveals" its syntactic sugar for the local
function, it does not use the "naive" way. Instead, define
division as shown below:
`local function foo ( <params> ) <body> end` ,
goes into
`local foo; foo = function ( <params> ) <body> end`
Therefore, we can safely use this syntax to re-
italic functions.
Of course, this trick won't work unless you have direct recursion -
the two functions call each other. In such cases, you need to explicitly write
put the appropriate descriptions of local variables:
```
local f, g - described local variables
function g ()
```

<*some code*> f () <*some code*>
end
function f ()
<*some code*> g () <*some code*>
end
In this example, the function f can not write a local function f ,
because in such a case Lua will create a new local variable-
the new f , leaving the old (referenced by g) uninitialized-
Noah.

6.3. Tail optimization calls

Another interesting feature of functions in Lua is that
Lua performs tail call optimizations. (It means that
Lua supports *tail recursion* optimization ,
although it is not directly related to recursion here, see exercise-
(see section 6.3.)
The tail call is actually a **goto** that looks like
function call. A tail call happens when a function is called
has another function as its last action. For example, in the following
In the following code, the call to the function g is tail:
function f (x) return g (x) end
After f calls g , it has nothing else to do. In similar
situations, the program does not need to return to the calling
function when the nested call has completed. Therefore after
tail call the program does not need to store any
information about the calling function on the stack. When the call to g ends
it seems that control goes directly to the point where it was
named f . Some language implementations, such as the Lua interpreter,
use this fact and do not allocate additional stack space
for tail call. We say that these implementations are supported
by *eliminating tail calls* (tail-call elimination).
Since tail calls do not use stack space,
number of nested tail calls that the program can
execute, it is simply not limited by anything. For example, we can you-
call the next function, passing any number as an argument
cop:

```
function foo (n)
if n> 0 then return foo (n - 1) end
end
```
This call will never result in a stack overflow.
The subtle point in eliminating tail calls is
the question of what is a tail call. For some
quite obvious candidates require that a defiant
the function does nothing else after the call, it is not executed. On-
example, in the following code, the function call g is not tail.
```
function f (x) g (x) end
```
The problem in this example is that after calling g, the function f
should discard the results of g before returning. Similarly, all the following
the next calls also do not satisfy the condition:
```
return g (x) + 1 - addition is required
return x or g (x) - must be converted to 1 value
return (g (x))
- must be converted to 1 value
```
In Lua, only a call like return *func* (*args*) is tail.
However, both *func* and its arguments can be complex expressions,
as Lua will execute them before calling. For example the following
the call is tailored:
```
return x [i] .foo (x [j] + a * b, i + j)
```

Exercises

Exercise 6.1. Write a function integral that gets
function f and returns an approximate value of its integral
rala.

Exercise 6.2. In Exercise 3.3, you had to write a function
the function that receives the polynomial (represented by the table)
and the value of the variable and returns the value of the polynomial for
this variable. Write a function that receives a lot of
goclen and returns a function that, when called for
any x value , will return the polynomial value for this
x . For instance:
```
f = newpoly ({3, 0, 1})
print (f (0))
-> 1
print (f (5))
-> 76
```

```
print (f (10))
-> 301
```

Exercise 6.3. Sometimes the language supporting optimization tail calls is called *supporting tail recursion* (properly tail recursive), if the optimization of tailings out calls is only supported for recursive calls.
wwii (Without recursive calls, the maximum call depth is wow is statically defined.)
Show that this is not true in a language like Lua: write a program that implements unlimited call depth wow without using recursion. (*Hint* : see section 8.1.)

Exercise 6.4. As we have seen, the tail call is a a clipped **goto** . Using this idea, rewrite the code for the maze games from section 4.4 using tail calls. Each block should become a new function and each **goto** becomes a tail call.

CHAPTER 7
Iterators and the generalized for

In this chapter, we will show you how to write iterators for generalized **for** (general **for** operator). Starting with simple iterators, we learn how to use the full power of generic for to write simpler and more efficient iterators.

7.1. Iterators and Closures

An iterator is any construct that allows you to iterate over
host elements of the set. In Lua, we usually represent iterators by
help functions: every time we call a function, it will
rotates the "next" item from the set.
Any iterator has to save its state somewhere between
challenges to know where he is and how to proceed.
Closures are an excellent mechanism for this task.
Recall that a closure is a function that refers to one
noah or several local variables from your environment.
These variables retain their values between successive
calls to the closure, thereby helping the closure understand where
it is on its way. Of course, to create a new closure
we also have to create non-local variables for it. Therefore,
the structure of a closure usually includes two functions at once: itself
closure and *factory* , a function that creates a closure along with
surrounding variables.
As an example, let's write a simple iterator for a list.
Unlike ipairs , this iterator will not return the index of every
the same element, but only its value:

```
function values (t)
local i = 0
return function () i = i + 1; return t [i] end
end
```

In this example, values is a factory. Every time we call
We use this factory, it creates a new closure (iterator). This is
The kaney stores its state in its external variables t and i . Each
Every time we call this iterator, it returns the following
value from list t . After the last element, the iterator will return **nil** ,
which marks the end of iterations.
We can use this iterator in a **while loop** :

```
t = {10, 20, 30}
iter = values (t)
- create an iterator
while true do
local element = iter ()
- call the iterator
if element == nil then break end
```

```
print (element)
end
```

However, it is much easier to use the generic **for** statement . After all, it was created for exactly this kind of iteration- niya:

```
t = {10, 20, 30}
for element in values (t) do
print (element)
end
```

The generic **for** does all the behind-the-scenes work for the iteration rationing: it stores the iterating function inside, so we don't needs a variable iter , it calls the iterator for each new iteration walkie-talkie, and it completes the iteration when the iterator returns **nil** . (We'll see in the next section that the generic **for** does even more than that.)

As a more advanced example, see Listing 7.1 for re- fetch all words from the current input file. For such a search, we two values are needed: the content of the current line (variable line) and where we are inside this line (variable pos). With these data, we can always generate the next word. Basic

The most important part of the iterating function is the call to string.find . This call searches for a word in the current line, starting at the current position. He described matches a 'word' using the pattern '% w +' , which one or more alphanumeric characters. If this call finds word, then the function updates the current position by the first character after a word and returns that word 1 . Otherwise, the iterator reads the following- string and repeats the search. If there are no more rows, it returns **nil** to signal the end of the traversal.

Despite its complexity, using allwords is extremely easy then:

```
for word in allwords () do
print (word)
end
```

This is a typical situation with iterators: they may not be so easy to write, but they are easy to use. This is not a problem, much more often, end users who program in Lua do not write their own iterators, but use the iterators provided by application.

Listing 7.1. Iterator to iterate over all words from the input file

```
function allwords ()
```

```
local line = io.read ()
- current line
local pos = 1
- current position in the line
return function ()
- iterator function
while line do
-– repeat as long as there are lines
local s, e = string.find (line, “% w +”, pos)
if s then
-– have you found the word?
pos = e + 1
-– next position after word
return string.sub (line, s, e) -– return word
else
line = io.read () - word not found; trying the trail. string
pos = 1
- start at the beginning of the line
end
end
return nil - no more lines, end of traversal
end
end
```

7.2. Generic for semantics

One of the drawbacks of the iterators discussed above is that
that we need to create a new closure for initialization
each new cycle. For most cases, this is not
problem. For example, in the case of the allwords iterator, the creation price is
a single closure is incomparable to the cost of reading an entire file. but
in some situations this can be significant. Such
cases we can use the generic **for itself** to store
states. In this section, we will see what possibilities for storing
state is offered by the generic **for** .

We have seen that a generic **for** loop stores iterables during a loop.
function within itself. It actually stores three values:
iterated function, *unchangeable state* (invariant state) and
control variable . Now let's get down to the details.

The syntax **for a** generic **for is** shown below:

for <*var-list*> in <*exp-list*> do
<*body*>
end

Here *var-list* is a list of one or more re-
variables separated by commas, and *exp-list* is a list of one or
multiple expressions, also separated by commas. Often a list
expression consists of a single element, a call to the iterate factory
rators. In the following code, for example, the list of variables is k , v ,
and the list of expressions consists of a single element, pairs (t) :

```
for k, v in pairs (t) do print (k, v) end
```

Often, a variable list also consists of just one variable -
noah, as in the following loop:

```
for line in io.lines () do
io.write (line, "\ n")
end
```

We name the first variable in the *control variable list-*
noah . During the whole cycle, its value is not equal to **nil** , because when
it becomes **nil** , the loop ends.
The first thing a **for** loop does is compute the values expressed
ny following **in** . These expressions must give three meanings,
used by the **for** statement : iterating function, immutable
the state and initial value of the manipulated variable. As in
multiple assignment, only the last (or the only
nth) list item can give more than one value; and the number of these
values are reduced to three, excess values are discarded, together
then the missing **nil** 's are added . (When we use simple
iterators, the factory only returns the iterating function,
this invariant state and control variable get
the value is **nil** .)
After this initialization, **for** calls the iterating function with
two arguments: an invariant state and a control transfer
change. (From the point of view of the **for** statement , this is an invariant state
doesn't make any sense at all. The **for** statement only passes the value
reading the state from the initialization step to calling the iterating function
for) then assigns the values returned by the iterating
function, variables declared in the variable list. If a
the first value (assigned to the manipulated variable) is
nil , then the loop ends. Otherwise, **for** executes its body and calls again
an iterating function, repeating the process.
More precisely the construction of the view

```
for var_1, ..., var_n in <explist> do <block> end
```

equivalent to the following code:

```
do
```

```
local _f, _s, _var = <explist>
while true do
local var_1, ..., var_n = _f (_s, _var)
_var = var_1
if _var == nil then break end
<block>
end
end
```

So if our iterating function is *f* , the immutable
the state *s* and the initial state for the control variable is
a_0 , then the control variable will run through the following values
$a_1 = f(s, a_0)$, $a_2 = f(s, a_1)$, etc., until a_i equals **nil** .
If **for** has other variables, then they just get an extra
solid values returned by *f* .

7.3. Stateless iterators

As its name implies, such an iterator does not store any
any state. Therefore, we can use the same
a stateless iterator in many loops, thus avoiding creating-
new closures.
As we have seen, the **for** statement calls the iterating function
with two arguments: immutable state and control transfer
changeable. A stateless iterator builds the next element of the loop,
using only these two values. A typical example of this iteration is
the torus is ipairs , which iterates over all the elements of the array:

```
a = {“one”, “two”, “three”}
for i, v in ipairs (a) do
print (i, v)
end
```

The state of this iterator is the table we are iterating over
(an immutable state that retains its value throughout
loop), and the current index (control variable). And ipairs
(factory) and the iterator itself is very simple, we could write them to
Lua as follows:

```
local function iter (a, i)
i = i + 1
local v = a [i]
if v then
return i, v
end
end
```

```
function ipairs (a)
return iter, a, 0
end
```
When Lua is ipairs (a) for a cycle **for** , she gets three
values: an iterating function iter , a as an invariant state
value and zero as the initial value for the control
change. Lua then calls iter (a, 0) , which gives 1, a [1] (unless
a [1] is no longer **nil**). The next iteration calls iter (a, 1) , which
returns 2, a [2] , and so on up to the first **nil** element .
The pairs function , which iterates over all the elements in a table, looks like
ms, except that the iterating function is a function
next , which is a standard Lua function:
```
function pairs (t)
return next, t, nil
end
```
Calling next (t, k) , where k is the key of table t , returns the following -
key in the table in no particular order, and also associated with
with this key the value as the second return value. Call
next (t, nil) returns the first pair. When there are no more pairs, then next
returns **nil** .
Some people prefer to use next explicitly , avoiding calling
pairs :
```
for k, v in next, t do
<loop body>
end
```
Recall that for casts its list of expressions to three values.
niyam, which are next , t and **nil** ; this is exactly what
what happens when calling pairs .
An iterator for traversing a linked list is another interesting one
an example of a stateless iterator. (As we have already mentioned, the connected
lists are not common in Lua, but sometimes we need them.)
```
local function getnext (list, node)
if not node then
return list
else
return node.next
end
end
function traverse (list)
return getnext, list, nil
end
```
Here we use the beginning of the list as an invariant state
(second value returned by traverse) and the current node as

ve control variable. When the iterated function getnext
will be called the first time, node will be **nil** , and therefore the function
will return list as the first node. In subsequent calls, node will already be
is not **nil** and therefore the iterator will return node.next as expected.
As usual, using this iterator is extremely simple:

```
list = nil
for line in io.lines () do
list = {val = line, next = list}
end
for node in traverse (list) do
print (node.val)
end
```

7.4. Iterators with complex condition

Often an iterator needs to store more state than
is placed in the variables of the invariant state and the control
variable. The simplest solution is to use a
kaniy. An alternative solution would be to pack everything you need
an iterator into a table and use that table as invariant
state for the loop. Using a table, an iterator can store
as much data as he needs. Moreover, he can change these
data the way he wants. Although the state is the same all the time
table (therefore it is invariant), the contents of the table can be
run throughout the cycle. Since such iterators store everything
their data is in a state, they usually ignore the second argument,
provided by the generic **for** loop (loop variable).
As an example of this approach, we will rewrite the iterator
allwords , which bypasses all words in the input file. This time
we will store its state in a table with two fields: line and
pos .
The function that starts the loop is pretty simple. She must ver-
Introduce an iterating function and an initial state:

```
local iterator
- to be determined later
function allwords ()
local state = {line = io.read (), pos = 1}
return iterator, state
end
```

The iterator function does most of the work :

```
function iterator (state)
```

```
while state.line do
- repeat while there are lines
- looking for the next word
local s, e = string.find (state.line, “% w +”, state.pos)
if s then
- found the word?
- update position
state.pos = e + 1
return string.sub (state.line, s, e)
else - word not found
state.line = io.read () - trying the next line ...
state.pos = 1
-- ... first
end
end
return nil
- there are no more lines, we end the loop
end
```

Whenever possible, you should try to write an iteration-
stateless tori, those that keep their entire state in the
for loop variables . With them, you do not create new objects when
start the cycle. If this model does not fit, then you should ask
to create closures. Also, it's prettier, closure is usually
is more efficient as an iterator than a table:
First, it is cheaper to create a closure than a table; secondly, access
accessing non-local variables is faster than accessing table fields.
We will see another way to write iterators later, using
by using coroutines. This is the most powerful solution, but it
somewhat more expensive.

7.5. Genuine iterators (true iterarators)

The term "iterator" is somewhat inaccurate, since it actually iterates
It iterates not an iterator, but a **for** loop . Iterators only provide
consecutive values to iterate over. Maybe more
a good term would be "generator", but the term "iterator" is already
became widespread in languages like Java.
However, there is another way to construct iterators, where
iterators actually do iteration. When we use
we use such iterators, we don't write a loop; instead we just
then we call an iterator with an argument describing that the iterator should
to do at each iteration. More precisely, the iterator receives as
ve argument is a function that it calls inside its loop.
As an example, let's rewrite the allwords iterator again

using this approach:

```
function allwords (f)
for line in io.lines () do
for word in string.gmatch (line, “% w +”) do
f (word)
- call function
end
end
end
```

To use this iterator, we just have to provide
inject the body of the loop as a function. If we just want to print each
after a word, then we use print :

```
allwords (print)
```

Often an anonymous function is used as the loop body. On-
example, the following code snippet counts how many times the word "hello"
occurs in the file:

```
local count = 0
allwords (function (w)
if w == “hello” then count = count + 1 end
end)
print (count)
```

The same task written using iterators previously
the style looked at is not much different:

```
local count = 0
for w in allwords () do
if w == “hello” then count = count + 1 end
end
print (count)
```

Similar iterators were very popular in older versions
Lua, when the language did not yet have a **for** statement . How do they compare
with iterators of the previously discussed style? Both styles have a
roughly the same overhead: one function call per iteration
walkie-talkie. On the other hand, it is easier to write an iterator using
genuine iterators (although we can get the same ease
using coroutines). On the other hand, previously considered
the new style is more flexible. First, it allows two or more pa-
parallel iteration. (For example, consider the case of bypassing
two files at once, comparing them word by word.) Second, it
allows the use of **break** and **return** inside a loop. With authentic
return iterators **return** from anonymous function, but not
from the loop. Therefore, I usually use traditional (i.e.
seen earlier) iterators.

Exercises

Exercise 7.1. Write an iterator fromto such that you follow-
the two loops are equivalent:

```
for i in fromto (n, m)
<body>
end
for i = n, m
<body>
end
```

Can you implement this with an iterator without co-
standing?

Exercise 7.2. Add a step parameter to the previous exercise.
neniyu. Can you still implement this with
soup of a stateless iterator?

Exercise 7.3. Write a uniquewords iterator that can
rotates all words from the given file without repetitions.
(*Hint* : start with the allwords code in Listing 7.1; use
use a spreadsheet to store all the words you have already
zeros.)

Exercise 7.4. Write an iterator that returns everything
non-empty substrings of the given string. (You will need a function
ration string.sub .)

CHAPTER 8
Compilation, execution and errors

Although we call Lua an interpreted language, Lua is always pre-
compiles the source code into an intermediate form before ex-
filling. (In fact, many interpreted languages do
ditto.) Having a compile file may sound strange
in relation to an interpreted language like Lua. However, the main
the feature of interpreted languages is not that they are not compiled.
are copied, but what is possible (and easy) to execute the generated

code on the fly. We can say that having a function like dofile is
this is what allows Lua to be called an interpreted language.

8.1. Compilation

Earlier, we introduced dofile as a kind of primitive operation for
executing blocks of Lua code, but dofile is actually an aid
gatelnaya function: all the hard work is done LoadFile . Like
dofile , loadfile loads a block of Lua code from a file, but it doesn't
completes this block. Instead, it only compiles this block
and returns the compiled block as a function. Moreover, in
unlike dofile , loadfile does not throw an error, it just returns
error code so that we can handle these errors ourselves. We can
Let's define dofile like this:

```
function dofile (filename)
local f = assert (loadfile (filename))
return f ()
end
```

Note the use of the assert function in order to
to *cause errors* (raise error), if loadfile fulfills a
a mistake.

For simple tasks, dofile is convenient because it does all the work.
bot in one call. However, loadfile is more flexible. In case of error
loadfile returns **nil** and an error message, which allows us to
handle the error in a convenient way. Moreover, if we need
but execute the file several times, then we can call once
loadfile and call the function it returns several times. This
the approach is much cheaper than calling dofile multiple times , since
the file is compiled only once.

The load function is similar to loadfile , except that it
takes a block of code not from a file, but from line 1 . For example, consider the following
next line:

```
f = load ("i = i + 1")
```

After executing this code, f will be a function that executes-
em i = i + 1 when called:

```
i = 0
f (); print (i) -> 1
f (); print (i) -> 2
```

The load function is quite powerful and we have to use it with
caution. It is also an expensive feature (compared to some

alternatives) and can lead to code that is very heavy
lo understand. Before you use it, make sure there is no more
simple way to solve the problem.
If you want a quick and dirty dostring (i.e. load and
execute block), you can directly use the result
tat load :
load (s) ()
However, if there is at least one syntax error, then load
will return **nil** and the final error will be something like *"attempt to call a nil value "* . For clearer error handling use
assert :
assert (load (s)) ()
There is usually no point in calling the load function on a literal
(that is, an explicitly quoted string). For example, the following two
the lines are roughly equivalent:
f = load ("i = i + 1")
f = function () i = i + 1 end
However, the second line is much faster since Lua compiled
functions together with the surrounding block. On the first line, you-
the load call includes a separate compilation.
Since load does not compile lexically aware
actions, then the two lines considered may not be entirely equivalent
valence. To see the difference, let's change it slightly
example:
i = 32
local i = 0
f = load ("i = i + 1; print (i)")
g = function () i = i + 1; print (i) end
f () -> 33
g () -> 1
The g function works on the local variable i as expected,
however the function f works with the global i , since load is always
compiles its blocks in the global environment.
The most typical use of load is to execute
external code, that is, snippets of code coming from outside your
programs. For example, you may want to plot the function
the option specified by the user; the user enters the function code, and
you then use load to execute it. Pay
note that load expects to receive a block, that is, statements. If a
you want to evaluate the expression, then you can append to the beginning
return expressions , which will give you a statement that returns a value

given expression. Let's look at an example:

```
print "enter your expression:"
local l = io.read ()
local func = assert (load (“return“ .. l))
print (“the value of your expression is“ .. func ())
```

Since the function returned by load is a normal function,
you can call her many times:

```
print "enter function to be plotted (with variable 'x'):"
local l = io.read ()
local f = assert (load (“return“ .. l))
for i = 1, 20 do
x = i - global 'x' (to be visible from outside the block)
print (string.rep (“*”, f ()))
end
```

(The string.rep function repeats a string a specified number of times.)
We can also call the load function by passing it as ar-
gument *reading function* (reader function). Reading function
can return a block of code in parts; load calls this function before
it's time until it returns **nil** , indicating the end of the block. For example
measures, the following call is equivalent to loadfile :

```
f = load (io.lines (filename, “* L”))
```

As we will see in chapter 22, the challenge io.lines (filename, "* L") WHO-
rotates a function that, when called, returns the following
line from file 2 . So load will read the block from
file line by line. The next option is similar, but more efficient
effective:

```
f = load (io.lines (filename, 1024))
```

Here the iterator returned by io.lines reads in blocks at
1024 bytes.
Lua treats each independent block as the body of an ano
a variable function with a variable number of arguments. For instance,
load (“a = 1”) returns an analog of the following function:

```
function (...) a = 1 end
```

Like any other function, blocks can define their own locale.
variable variables:

```
f = load (“local a = 10; print (a + 20)”)
f () -> 30
```

Using these capabilities, we can rewrite our example with
plotting the graph so as not to use the global re-
variable x :

```
print "enter function to be plotted (with variable 'x'):"
local l = io.read ()
local f = assert (load (“local x = ...; return“ .. l))
```

```
for i = 1, 20 do
print (string.rep (“*”, f (i)))
end
```
We put the description “local x = ...” at the beginning of the block to define divide x as a local variable. When we call f with an argument ment i , this argument becomes the value of expression ... .
The load function never raises an error, in case of an error it just returns **nil** and an error message:
```
print (load (“ii”))
-> nil [string “ii”]: 1: '=' expected near 'i'
```
Moreover, these functions have no side effect. They only compile the block to an internal representation and return- give the result as an anonymous function. A common mistake is that it is assumed that block loading determines the function tions (defined in this block). In Lua, function definitions are assignment; and as such they happen at runtime, not at compile time. For example, let's say we have a file foo.lua with the following content:
```
- file 'foo.lua'
function foo (x)
print (x)
end
```
Then we run the command
```
f = loadfile (“foo.lua”)
```
After this command, foo has been compiled but not yet defined. To define it, we must execute a block:
```
print (foo) -> nil
f () - defines 'foo'
foo (“ok”) -> ok
```
In serious programs that need to execute external code, you have to handle all errors that occur when loading block. Moreover, you may want to start a new unit in the protection environment to avoid unpleasant side effects
Comrade We'll discuss environments in detail in Chapter 14.

8.2. Precompiled code

As I mentioned at the beginning of this chapter, Lua precompiles the running code before executing it. Lua also allows distribution take the code in precompiled form.
The simplest way to get a precompiled file is

also called a *binary block* in Lua - is the use of
the luac program included in the standard delivery. For instance,
next call creates prog.lc file with precompiled
version of the prog.lua file :

```
$ luac -o prog.lc prog.lua
```

The interpreter can then execute this file as a normal
Lua code, working in the same way as with the original file:

```
$ lua prog.lc
```

Lua allows for precompiled code pretty much wherever
it allows source code. In particular, loadfile and load take
the input is also precompiled code.
We can write a simple luac replacement directly to
Lua:

```
p = loadfile (arg [1])
f = io.open (arg [2], “wb”)
f: write (string.dump (p))
f: close ()
```

The main function here is string.dump : it gets the function
lua and returns its precompiled code as a string,
properly formatted to load it into Lua.
The luac program also introduces some interesting options.
In particular, the -l option prints a list of all opcodes that
the compiler generates for the given block. As an example, foxes
Thing 8.1 contains the output of the luac program launched with the -l option ,
for the following one line file:

```
a = x + y - z
```

(We will not discuss Lua internals in this book; if you
interested in information about these opcodes, then search the Internet
the words "lua opcode" will give you reasonably accurate information.)
Precompiled code is not always smaller than the original
code, but it loads faster. Another plus is that
this gives you protection against accidental changes to the source. However, in
deviation from source code, maliciously modified binary code
can crash the Lua interpreter or even execute
user-supplied machine code. At startup, the usual
There is nothing to worry about with this code. However, you should avoid
using
running untrusted code in precompiled form. The function
load has a special option for just this task.

Listing 8.1. Sample luac –l output

```
main <stdin: 0,0> (7 instructions, 28 bytes at 0x988cb30)
```

```
0+ params, 2 slots, 0 upvalues, 0 locals, 4 constants, 0 functions
1 [1] GETGLOBAL 0 -2; x
2 [1] GETGLOBAL 1 -3; y
3 [1] ADD 0 0 1
4 [1] GETGLOBAL 1 -4; z
5 [1] SUB 0 0 1
6 [1] SETGLOBAL 0 -1; a
7 [1] RETURN 0 1
```

In addition to the required first argument, load has three more optional natal arguments. The second argument is the name of the block that The swarm is only used in error messages. The fourth argument cop is the environment, we will look at it in detail in chapter 14. Third the argument is exactly what we are interested in now; he controls what types of blocks can be loaded. If this argument is is present, then it must be a string: the string “t” allows loading only text blocks, line “b” allows loading only binary (precompiled) blocks, and the string "bt" (the value of the the default) allows you to load blocks of both types.

8.3. C code

Unlike Lua code, C code must be forged with the app before using it. In a number of popular operating systems, the easiest way to do this is using the dynamic linking feature. However, given This feature is not part of the ANSI C specification; so no re- a wearable way to do this.

Lua usually does not include features that cannot be implemented in ANSI C. However, with dynamic linking, tion is different. We can consider it as the basis of all others opportunities: as soon as we have it, we can immediately load reap any feature that is not currently in Lua. Therefore, in this Otherwise, Lua abandons portability rules and implements dynamic linking for a number of platforms. Standard implementation zation offers this feature for Windows, Mac OS X, Linux, FreeBSD, Solaris and most other UNIX implementations. Transfer this capability to other platforms doesn't have to be difficult; refer to your distribution. (To check this, run print (package.loadlib (“a”, “b”)) from the Lua command line and then look at the result. If it reports a file that does not exist, then

you have dynamic linking support. Otherwise
an error message will tell you that this feature is not supported
or not installed.)
Lua provides all the dynamic linking capabilities
through one function, package.lodlib . This function receives two
string arguments: full library path and function name from
this library. Therefore, her typical call looks like
but below:

```
local path = "/usr/local/lib/lua/5.1/socket.so"
local f = package.loadlib (path, "luaopen_socket")
```

The loadlib function loads the specified library and connects
her to Lua. However, it does not call the specified function. Instead he
returns a C function as a Lua function. In case of an error when
loading a library or finding an initializing function
loadlib returns **nil** and an error message.
The loadlib function is very low level. We must re-
Pass the full path to the library and the correct function name (including
teas, beginning underscores added by the compiler). Often we
load libraries to C using require . This function looks for
library and uses loadlib to load the initialization
functions for the library. When called, this initializing function
ration builds and returns a table with functions from this library, like
does the usual Lua library. We will discuss require in section 15.1 and
See section 27.3 for more information on C libraries.

8.4. Errors

Errare humanum est [3] . Therefore, we must handle errors like this
good as we can. Since Lua is an extension language,
often embedded in an application, we cannot just fall or
exit if an error occurs. Instead, when
yes an error occurs, Lua interrupts the execution of the current block and
returns control to the application.
Any unexpected situation Lua encounters will cause
leads to *call errors* (raises an error). Errors occur when
you (more precisely, your program) cannot add values that
are not numbers, do not index a table, etc. (You can
want to change this behavior using *metatables* as we will see

later.) You can also explicitly cause errors by calling
error function with an error message as an argument. Usually
this function is the correct way to report a bug in
your code:

```
print "enter a number:"
n = io.read (“* n”)
if not n then error (“invalid input”) end
```


Humans tend to make mistakes. (*lat* .)
This way of calling error is so common that for this
Lua has a built-in assert function :

```
print "enter a number:"
n = assert (io.read (“* n”), “invalid input”)
```

The assert function checks if its first argument is valid
is not false, and simply returns this argument; if the argument is false, then
assert raises an error. Its second argument, the error message,
not required. However, keep in mind that assert is a normal function.
tion. As with all functions, Lua always computes before calling it.
passes her arguments. So if you write something like

```
n = io.read ()
assert (tonumber (n), “invalid input:“ .. n .. “is not a number”)
```

then Lua will always do the concatenation, even if n is a number. Therefore-
In such cases, it may be better to use an explicit test.
When a function detects an unexpected situation (*excluding*
reading), it can go in two ways: return an error code (usually
nil) or raise an error using error . There are no tough
rules for choosing between these two options, but we can
offer general advice: an exception that's easy to get around,
should throw an error; otherwise, an error code should be returned.
For example, let's consider the sin function . How should she behave
lead if its argument is a table? Suppose it is possible
rotates the error code. If we need to check for errors, then we
we can write something like

```
local res = math.sin (x)
if not res then
```

```
- error?
<error-handling code>
```
However, we can easily change this exception *before* calling
functions:
```
if not tonumber (x) then
- x is not a number?
<error-handling code>
```
Often we do not check for either the argument or the result of calling sin , if the argument is not a number, it means that something is wrong in our program. In a similar situation, stop computing and call error is the simplest and most practical way to handle this exception.

On the other hand, let's look at the io.open function , which Paradise opens the file. How to behave if asked to open non-existent file? In this case, there is no simple way Soba to check for a call for an exception before calling this function tion. On many systems, the only way to check that file exists, is to try to open it. So if io.open cannot open the file for some external reason (for- example, "file does not exist" or "no rights!"), then it just might rotates **nil** along with the error message. In this case, you have a chance to handle the situation in an appropriate way, such as asking with a different filename:

```
local file, msg
repeat
print "enter a file name:"
local name = io.read ()
if not name then return end — nothing was entered
file, msg = io.open (name, “r”)
if not file then print (msg) end
until file
```

If you do not want to handle a similar situation, but do not want to be safe anyway, you can simply use use assert :

```
file = assert (io.open (name, “r”))
```

This is a typical idiom for Lua: if io.open completed with The error coy, then assert will raise an error.

```
file = assert (io.open (“no-file”, “r”))
-> stdin: 1: no-file: No such file or directory
```

Note how the error message that is the second result of io.open turns out to be the second argument when call to assert .

8.5. Error processing and exceptions

For many applications, you do not need to do any processing.
a lot of errors in Lua; all processing is done by the application itself. All work
Lua begins by being called by an application, usually consisting of
block execution. If an error occurs, then this call returns
error code and the application can respond appropriately
bed. In the case of a separate interpreter, its main loop is simply
prints an error message and continues.
However, if you need to handle errors in Lua, then you should
use pcall *function* (protected call) to encapsulate your
th code.
Let's say you want to execute a piece of Lua code and catch
any error that occurs while executing it. Your first
the step is to wrap this piece of code in a function; quite an hour
then anonymous functions are used for this. Then you call
this function using pcall :

```
local ok, msg = pcall (function ()
<some code>
if unexpected_condition then error () end
<some code>
print (a [i]) — possible error: 'a' may not be a table
<some code>
end)
if ok then — no error occurred while executing the protected code
<regular code>
else
— the protected code caused an error; process her
<error-handling code>
end
```

The pcall call invokes its first argument in *protected mode*
me , so that all errors during the execution of the function are caught.
cts. If there are no errors, then the pcall call returns **true** and
all values returned by the function. Otherwise, it returns **false** and
error message.
Despite its name, the error message need not be
by line: calling pcall will return whatever Lua value you pass-
whether error .

```
local status, err = pcall (function () error ({code = 121}) end)
print (err.code) -> 121
```

These mechanisms provide everything you need to process the data.
key in Lua. We throw an exception with error and

intercept it using pcall . The error message is
specifies the type of error.

8.6. Error messages and call stack

Although we can use the sign as an error message
any type, usually error messages are strings that describe
wondering what went wrong. In the event of an internal
errors (e.g. trying to index a non-table) message
Lua generates an error; otherwise the error message becomes
the value passed to the error function . When the error message
is a string, then Lua tries to add some information
about the place where the error occurred:

```
local status, err = pcall (function () a = “a” +1 end)
print (err)
-> stdin: 1: attempt to perform arithmetic on a string value
local status, err = pcall (function () error (“my error”) end)
print (err)
-> stdin: 1: my error
```

The error message contains the file name (in the example it is stdin)
and the line number in it (in the example it is 1).
The error function has a second optional parameter which
tells the *level* where to report the error; you are using this pas-
a yardstick to blame someone else for a mistake. For instance,
you wrote a function that immediately checks that it was
duly called:

```
function foo (str)
if type (str) ~ = “string” then
error (“string expected”)
end
<regular code>
end
```

Then someone calls your function with the wrong argument-
Tom:

```
foo ({x = 1})
```

In this case, Lua points to your function - after all,
it was she who caused the error - not the real culprit, the one who
called her with the wrong argument. In order to fix this,
we can pass error that the error you are reporting is
Nick at level 2 in the call stack (level 1 is your function):

```
function foo (str)
```

```
if type (str) ~ = “string” then
error (“string expected”, 2)
end
<regular code>
end
```

Often when an error occurs, we want to get more
more accurate information than just where it originated. How mi-
At least we want the call stack that resulted in the error. When pcall
returns an error, it destroys part of the stack (part
from it until the error occurs). Accordingly, if we
want to get a call stack, then we must build it before
gates from pcall . Lua provides the xpcall function for this . Cro-
the function to be called receives a second argument,
error handling function . In case of a Lua error
calls this error handling function before flushing the stack,
therefore, she can use the debug library to obtain
any additional information about the error. The two most
common error handlers are debug.debug ,
giving you a command line in Lua so you can do it yourself
see what was happening when the error occurred; and
debug.traceback which builds an extended error message
ke, including the call stack [4] .
It is the latter function that the independent inter-
pretator for printing error messages.

Exercises

Exercise 8.1. It is often necessary to add code to the beginning of the download.
pressed block. (We have already seen an example in this chapter when we
added code to **return** .) Write a function loadwithprefix ,
which works like load , except that it pre-
prepends its additional argument to the beginning of the load
th block.
Like the original load function , loadwithprefix should
accept blocks represented both as strings and read-
functions. Even when the original unit
is a string, loadwithprefix should not be explicitly concatenated
string the passed argument with a block. Instead, she

should call load with the appropriate read function-
it, which first returns the passed argument, and then -
block.

Exercise 8.2. Write a multiload function that generalizes
schaet loadwithprefix , receiving the input list reading function
tions, as in the following example:

4
In Chapter 24, we will learn more about these functions and the debug library.

```
f = multiload (“local x = 10;”,
io.lines (“temp”, “* L”),
“Print (x)”)
```

For the above example, multiload should load
block equivalent to concatenating the string “local ...” with co-
by holding the temp file and the "print (x)" line . Like function
loadwithprefix , this function itself is nothing concatenated
should not.

Exercise 8.3. The string.rep function in Listing 8.2 is used
zuet *algorithm binary multiplication* (binary multiplication
algorithm) to concatenate n copies of the given string s . For
any fixed n, we can create a specialized
bathroom version of string.rep , expanding the loop into a sequential
the number of commands r = r..s and s = s..s . As an example for n = 5
we get the following function:

```
function stringrep_5 (s)
local r = “”
r = r .. s
s = s .. s
s = s .. s
r = r .. s
return r
end
```

Write a function that, for a given n, returns
stringrep_n function . Instead of using a closure
your function should build the function text in Lua with
appropriate commands r = r..s and s = s..s and then use
use load to get the final function. Compare-
those are the performance of the string.rep function and the one you got
functions.

Exercise 8.4. Can you find a value for p such that
pcall expression (pcall, f) will return **false** as the first value
nie?

Chapter 9
Coroutines

A coroutine is like a thread (in the multi-threaded sense): it is a thread execution with its stack, its local variables and its *instruction pointer* (instruction pointer); but he shares the global-point variables and pretty much everything else with other coroutines-mi. The main difference between threads and coroutines is that a multi-threaded program executes all of these threads in parallel but. Coroutines work together: at any given time, gram executes only one of its coroutines, and this one executes May, a coroutine suspends its execution only when will explicitly ask for it.

Coroutines are a very powerful concept. And so many of their applications are quite complex. Don't worry if you don't understand some of the examples in this chapter on first reading. You can read to the end of the book and come back later. But please, come back it will be a well spent time.

9.1. Fundamentals of coroutines

Lua keeps all the functions for working with coroutines in a table `coroutine` . The `create` function creates new coroutines. She has everything-Its one argument is the function that the coroutine will execute. It returns a value of type `thread` , which is the created coroutine. Often the `create` argument is anonymous function as below:

```
co = coroutine.create (function () print (“hi”) end)
print (co) -> thread: 0x8071d98
```

A coroutine can be in one of four states: prio-suspended, running, dead, and normal. We can find out the state of the coroutine at using the `status` function :

```
print (coroutine.status (co)) -> suspended
```

When we create a coroutine, it starts in a suspended state. nom condition; coroutine doesn't start automatically executing

our body when we create it. The coroutine.resume function continues
starts (starts) execution of the coroutine, changing its state from
suspended in running:

```
coroutine.resume (co) -> hi
```

In this first example, the coroutine body simply prints "hi"
and terminates execution, leaving the coroutine in destroyed
condition:

```
print (coroutine.status (co)) -> dead
```

Until now, coroutines have looked like just a complicated way
function calls. The real power of coroutines comes from function
yield , which allows a running coroutine to pause
its execution so that it can be continued later. let's
consider a simple example:

```
co = coroutine.create (function ()
for i = 1, 10 do
print ("co", i)
coroutine.yield ()
end
end)
```

Now, when we continue with this function, it will
repairs its execution and executes before the first yield :

```
coroutine.resume (co) -> co 1
```

If we now check its status, we will see that this co-
the program is paused and, therefore, we can
continue its execution:

```
print (coroutine.status (co)) -> suspended
```

From the point of view of the coroutine, all the activity that occurs
dit while the coroutine is suspended happens inside the call
yield . When we continue to execute the coroutine, it returns
comes from the yield call and continues its execution until the next
th call yield or the end of the coroutine:

```
coroutine.resume (co) -> co 2
coroutine.resume (co) -> co 3
...
coroutine.resume (co) -> co 10
coroutine.resume (co) - prints nothing
```

During the last call to resume, the loop ends and ends
Execute the function without printing anything. If we try
resume its execution again, then resume will return **false** and the message
about the error:

```
print (coroutine.resume (co))
-> false cannot resume dead coroutine
```

Note that resume executes the coroutine body in
protected mode. Therefore, in the event of any
errors while executing a coroutine Lua will not show a message
error message, but will simply return control to the call to resume .
When a coroutine continues executing another coroutine,
then it is not suspended; in the end we cannot continue
live her fulfillment. However, it is not executable, because
how many coroutines being executed is another coroutine.
Therefore, her status is called *normal* .
A nice feature in Lua is that the resume pair -
yield can exchange data. The first call to resume (which
there is no yield call waiting for it) passes its additional
arguments of the main coroutine function:

```
co = coroutine.create (function (a, b, c)
print ("co", a, b, c + 2)
end)
coroutine.resume (co, 1, 2, 3) -> co 1 2 5
```

Calling resume returns after **true** , indicating that there is no error-
side, all arguments passed to the yield call :

```
co = coroutine.create (function (a, b)
coroutine.yield (a + b, a - b)
end)
print (coroutine.resume (co, 20, 10)) -> true 30 10
```

Similarly, yield returns all arguments passed according to
The corresponding call to resume :

```
co = coroutine.create (function (x)
print ("co1", x)
print ("co2", coroutine.yield ())
end)
coroutine.resume (co, "hi") -> co1 hi
coroutine.resume (co, 4, 5) -> co2 4 5
```

Finally, when the coroutine finishes its execution, everything is
values returned by its main function are passed as a result
resume :

```
co = coroutine.create (function ()
return 6, 7
end)
print (coroutine.resume (co)) -> true 6 7
```

We usually rarely use all of these features in one and the same
the same coroutine, but they all have their own uses.
For those who already know something about coroutines it is important to clarify
some concepts before we move on. Lua offers
what is called *asymmetric coroutines* . It means that

it has a function to pause the execution of a coroutine and
another function to continue execution of the suspended
coroutines. Some languages have *symmetric coroutines* ,
when there is only one function to transfer control from one
coroutines are different.
Asymmetric coroutines are called semi-coroutines by some.
mami (not being symmetrical, they are not co-). However, others
use the same term semi-routines to denote a limited
implementation of coroutines, where a coroutine can be suspended
do it only when it does not call any other
function, that is, when it has no pending calls. Other words
you, only the main body of such a coroutine can yield yield .
Generators in Python are examples of such semi- routines.
Unlike the difference between symmetrical and unbalanced-
mi coroutines, the difference between coroutines and generators
(as implemented in Python) is much deeper; generators just don't
powerful enough to implement some interesting con-
instructions that we can do with normal coroutines.
Lua offers complete non-symmetric coroutines. Those who
prefer symmetric coroutines, can implement them in
based on the asymmetric capabilities of Lua. This is not difficult. (Fact-
every control transfer performs a yield , followed by
blows resume .)

9.2. Channels and filters

One of the most important use cases for coroutines
is the task of the producer-consumer.
Let's pretend we have a function that is constantly running
reads values (for example, reads them from a file), and another function,
which constantly consumes these values (for example, writes to another
goy file). Typically these two functions look like this:

```
function producer ()
while true do
local x = io.read () - produce new value
send (x)
- send it to the consumer
end
end
function consumer ()
```

```
while true do
local x = receive () - get value from producer
io.write (x, "\ n")
- consume it
end
end
```

(In this implementation, both producer and consumer are
forever. However, they can be easily changed to stop when more
no data.) The challenge here is to connect you-
calls send and receive . This is a typical example of the problem "who has the main
cycle". Both the producer and the consumer are active, each has its own
main loops, and each assumes that the other is a call
my service. For this particular example, one can easily change
structure of one of the functions, expanding its cycle and making it passive
noah side. However, in real cases, such a change can
to be far from easy.

Coroutines provide an ideal mechanism for connecting
producer and consumer, since the resume – yield pair
reverses the usual relationship between caller and caller
removable. When a coroutine calls yield , it does not call a new one
function; instead, it returns control from the current call.
wa (resume). Likewise, calling resume does not start a new function,
and terminates the call to yield . This is exactly what we need to connect
neniya send and receive , so that each acts as
as if he is the main one, and the second is subordinate.

Therefore, receive continues to execute the producer, so it
can produce a new value; and send returns this value
back to the consumer:

```
function receive ()
local status, value = coroutine.resume (producer)
return value
end
function send (x)
coroutine.yield (x)
end
```

Of course, the producer must also be a coroutine:

```
producer = coroutine.create (
function ()
while true do
local x = io.read () — produce a new value
send (x)
end
end)
```

With this design, the program starts by calling the consumer. Kog-
yes, the consumer needs value, he resumes production
body, which is executed until it has a ready
value that it conveys to the consumer and does not stop until
until the consumer continues executing again. So
thus we get what is called *sweat-driven* design
fighter an (consumer-driven). Another option would be to write
program using a *manufacturer-driven* design where sweat-
a child is a coroutine.
We can extend this design with filters that
are tasks between the manufacturer and the
rebel and performing data conversion. *The filter* is
producer and consumer at the same time, therefore he
owes producer execution to get new value
and uses yield to pass this value to the consumer. In ka-
as a simple example, we can add to our previous
code a filter that inserts a line number at the beginning of each line.
The code is shown in Listing 9.1. In the end, we just need to create a compo
nents, connect them and start executing the final consumer:

```
p = producer ()
f = filter (p)
consumer (f)
```

Or even simpler:

```
consumer (filter (producer ()))
```

Listing 9.1. Consumer and manufacturer with filters

```
function receive (prod)
local status, value = coroutine.resume (prod)
return value
end
function send (x)
coroutine.yield (x)
end
function producer ()
return coroutine.create (function ()
while true do
local x = io.read () - produce new value
send (x)
end
end)
end
function filter (prod)
return coroutine.create (function ()
for line = 1, math.huge do
local x = receive (prod) - get new value
```

```
x = string.format (“% 5d% s”, line, x)
send (x)
- send it to the consumer
end
end)
end
function consumer (prod)
while true do
local x = receive (prod) - get new value
io.write (x, “\ n”)
- consume new value
end
end
```

Listing 9.2. Function to get all permutations of n elements a

```
function permgen (a, n)
n = n or #a - default 'n' is the size of 'a'
if n <= 1 then - nothing to do?
printResult (a)
else
for i = 1, n do
- put the i-th element at the end
a [n], a [i] = a [i], a [n]
- create all permutations of other elements
permgen (a, n - 1)
- restore the i-th element
a [n], a [i] = a [i], a [n]
end
end
end
```

If you've thought about pipes in UNIX, then you're not alone. After all, coroutines are a variant of *non-preemptive multi-tasks* (non-preemptive multitasking). With channels every task runs as a separate process; with coroutines each task runs as a separate coroutine. Channels provide bu-fer between the writer (producer) and the reader (consumer), therefore, some freedom in their relative speeds is possible. This is important for the channel, since the cost of switching between processes themselves are high. With coroutines, the cost of switching between tasks much smaller (like a function call), so writing and the reader can go toe-to-toe.

9.3. Coroutines like iterators

We can consider loop iterators as a separate example producer-consumer: the iterator produces values that consumed by the body of the cycle. Therefore, it is quite natural

use coroutines to write iterators. Really-
but, coroutines are a powerful tool for this purpose.
Again, the key feature is their ability to flip
the relationship between caller and callee. With this feature
we can write iterators without worrying about keeping state between
by successive calls to the iterator.
To illustrate this use case,
let's write an iterator to iterate over all permutations of a given
array. Writing an iterator like this isn't easy, but pretty
just write a recursive function that builds all these overrides
new. The idea is simple: put each element at the end in turn
array and recursively generate any remaining permutations.
The code is shown in Listing 9.2. In order for it to work, we must
we write the corresponding function printResult and call
permgen with proper arguments:

```
function printResult (a)
for i = 1, #a do
io.write (a [i], ““)
end
io.write (“\ n”)
end
permgen ({1,2,3,4})
-> 2 3 4 1
-> 3 2 4 1
-> 3 4 2 1
...
-> 2 1 3 4
-> 1 2 3 4
```

Once the generator is ready, it is very easy to convert it to
iterator. First, we'll replace printResult with yield :

```
function permgen (a, n)
n = n or #a
if n <= 1 then
coroutine.yield (a)
else
<as before>
```

Then we define a factory that will run the generator
inside a coroutine, and create an iterating function. For semi-
the next permutation, the iterator simply continues
nenie coroutine:

```
function permutations (a)
local co = coroutine.create (function () permgen (a) end)
return function () - iterator
local code, res = coroutine.resume (co)
```

```
return res
end
end
```
After that we can easily iterate over all permutations of the array.
using the **for** statement :
```
for p in permutations {“a”, “b”, “c”} do
printResult (p)
end
-> bca
-> cba
-> cab
-> acb
-> bac
-> abc
```
The permutations function uses a typical Lua pattern,
which hides the resume of the coroutine inside the function. This
the pattern is so common that Lua provides special
function for it: coroutine.wrap . Like create , wrap creates
new coroutine. Unlike create , wrap does not return itself.
coroutine; instead, it returns a function that when
the call continues execution of this coroutine. Unlike
resume , it does not return the error code as the first value; together
then it causes an error. Using wrap , we can write
permutations as follows:

```
function permutations (a)
return coroutine.wrap (function () permgen (a) end)
end
```
It is generally easier to use coroutine.wrap than coroutine.
create . It gives us exactly what we need from a coroutine:
function to continue its execution. However, it is less flexible
kaya. There is no way to check the status of a coroutine created-
noah with wrap . Moreover, we cannot check for errors.
at run time.

9.4. Non-displacing multi-threading

As we saw earlier, coroutines provide an option for
noisy multi-threading. Every coroutine is equivalent to a thread. Couple
yield-resume switches control from one thread to another thread.
However, unlike ordinary multi-threading, a coroutine is not
are preemptive. While the coroutine is running,
it cannot be stopped from the outside. She interrupts her execution

only when it explicitly requests it (via a call to yield). For
In some applications, this is not a problem, rather the opposite. Pro-
gramming is much easier in the absence of displacement. You do not
you need to worry about sync errors because all sync
chronization is obvious. You just need to make sure that the coroutine
ma calls yield outside the critical area of the code.
However, with non-displacing multi-threading, as soon as some
the thread calls a blocking operation, the whole program is blocked
until this operation completes. For most,
this is unacceptable, which leads to the fact that many programmers
you don't see coroutines as an alternative to traditional
multi-threading. As we will see here, there is an interest in this problem.
a new (and obvious, moreover) solution.
Let's look at a typical multi-threaded task: we want
download multiple files over HTTP. To download several
of these files, we first need to figure out how to download one file.
In this example, we will look at the bib-designed by Diego Nehab
the LuaSocket *library* . In order to download a file, you must first install
connect to the site containing this file, get the file
(in blocks) and close the connection. In Lua, we can write this next
in a blowing manner. First, we load the LuaSocket library:

```
local socket = require “socket”
```

Then we define the site and the file we want to download. In that
for example, we will download the HTML 3.2 reference manual from the site
World Wide Web Consortium:

```
host = “www.w3.org”
file = “/TR/REC-html32.html”
```

Then we open a TCP connection to port 80 (standard
port for HTTP connections) of this site:

```
c = assert (socket.connect (host, 80))
```

This operation returns the connection object that we are using
we use to send a request to receive a file:

```
c: send (“GET“ .. file .. “HTTP / 1.0 \ r \ n \ r \ n”)
```

We then read the file in 1 KB blocks, writing each block to
standard output:

```
while true do
local s, status, partial = c: receive (2 ^ 10)
io.write (s or partial)
if status == “closed” then break end
end
```

The receive function returns either the string it read, or

nil on error; in the latter case, it also returns the code
errors (status) and what she read before the error (partial). When the site
closes the connection, we print the remaining data and exit
from the loop.
After downloading the file, we close the connection:
c: close ()
Now that we know how to download one file, let's go back to
the problem of downloading multiple files. The simplest approach is to
Children download one file at a time. However, this consistent sub-
the move when we start reading the file only after we finish
with the previous file is too slow. When reading a file by
the network, the program spends most of its time waiting for data.
More precisely, she spends most of her time blocked
in the receive call . Therefore, the program can be executed significantly
It is much faster if it downloads all files at once. Then when
the connection has no ready data, the program can read the data
from another connection. It is clear that coroutines provide convenience
a good way to organize these simultaneous downloads. we
create a new thread for each downloaded file. When the thread
there is no ready data, it transfers control to the dispatcher, who
calls another thread.
In order to rewrite the program using co-
programs, we first need to rewrite the downloading code as
function. The result is shown in Listing 9.3. Since we are not interested
resche the contents of the file, the function reads and prints the file size
instead of writing the file to standard output (when we have several
they read several files at once, the output would be complete
mishmash).

Listing 9.3. Code for downloading a page from the network

```
function download (host, file)
local c = assert (socket.connect (host, 80))
local count = 0
- counts number of bytes read
c: send (“GET“ .. file .. “HTTP / 1.0 \ r \ n \ r \ n”)
while true do
local s, status = receive (c)
count = count + #s
if status == “closed” then break end
end
c: close ()
print (file, count)
end
```

In the resulting code, we use the helper function
(receive) to receive data from the connection. With successive
However, the code would look like this:

```
function receive (connection)
local s, status, partial = connection: receive (2 ^ 10)
return s or partial, status
end
```

For parallel implementation, this function must receive data
without blocking. Instead, if the required data is not available,
then it calls yield . The new code looks like this:

```
function receive (connection)
connection: settimeout (0)
— do not block
local s, status, partial = connection: receive (2 ^ 10)
if status == “timeout” then
coroutine.yield (connection)
end
return s or partial, status
end
```

The settimeout (0) call does any operation on the connection.
blocking. When the status of the operation is “timeout” , it means that
the operation completed without completing the request. In this case, the thread
is
Give control to another thread. The non- **false** argument passed to
ny yield statement , informs the dispatcher that this thread still performs
your task. Please note that even in the case of the “timeout” status
the partial variable still contains the read data.
Listing 9.4. contains dispatcher code and additional code.
The threads table contains a list of all active threads for the disk.
petcher. The get function ensures that each downloaded file
downloaded in a separate thread. The dispatcher itself is actually
just a loop that iterates over all the threads, starting them to execute-
one by one. It also removes from the list those threads that
have already completed the download. The cycle stops when no more
remains of threads.

Listing 9.4. Dispatcher

```
threads = {}
— list of all running threads
function get (host, file)
- create a coroutine
local co = coroutine.create (function ()
download (host, file)
```

```
end)
- insert it into the list
table.insert (threads, co)
end
function dispatch ()
local i = 1
while true do
if threads [i] == nil then
— no more threads?
if threads [1] == nil then break end — is the list empty?
i = 1 - restart the loop
end
local status, res = coroutine.resume (threads [i])
if not res then
— has the thread finished downloading?
table.remove (threads, i)
else
i = i + 1
— go to the next thread
end
end
end
```

Finally, the main routine creates the required threads and calls
there is a dispatcher. For example, to download four documents from the site
W3C, the main program might look like below:

```
Non-displacing multi-threading
host = “www.w3.org”
get (host, “/TR/html401/html40.txt”)
get (host, “/TR/2002/REC-xhtml1-20020801/xhtml1.pdf”)
get (host, “/TR/REC-html32.html”)
get (host, “/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.txt”)
dispatch () - main loop
```

On my computer, downloading these four files using
Coroutine processing takes 6 seconds. With sequential download
it takes more than twice (15 seconds).
Despite this optimization, this latest implementation is still
far from optimal. Everything works well as long as though
one thread would have something to read. However, when no thread has
ready to read data, the dispatcher constantly switches from
threads to thread just to make sure there are no ready-made
data. As a result, this implementation takes almost 30 times more
CPU time than the serial version.
In order to avoid this situation, we can use
call the select function from the LuaSocket library: it allows you to

block a program that is pending while changing
status in the connection group. Implementation changes are minor
us: we only need to change the dispatcher as shown in the listing
9.5. In a loop, the new dispatcher collects connections in the timedout table .
for which there is no ready-made data. (Remember that receive pe-
gives similar joins to the yield function , so again
running them). If none of the connections have ready data, then
the dispatcher calls select to wait for at least one
One of these connections will change the status. This final re-
Alization works as fast as the previous one. However, she
uses only slightly more CPU time than the serial
naya implementation.

Listing 9.5. Dispatcher using select

```
function dispatch ()
local i = 1
local timedout = {}
while true do
if threads [i] == nil then — no more threads?
if threads [1] == nil then break end
i = 1
— start the cycle over again
timedout = {}
end
local status, res = coroutine.resume (threads [i])
if not res then
— has the thread finished its work?
table.remove (threads, i)
else
— waiting time out
i = i + 1
timedout [#timedout + 1] = res
if #timedout == #threads then — are all threads blocked?
socket.select (timedout)
end
end
end
end
```

Exercises

Exercise 9.1. Use coroutines to pre-
form Exercise 5.4 into a generator for combinations that
ry can be used as follows:

```
for c in combinations ({“a”, “b”, “c”}, 2) do
printResult (c)
end
```

Exercise 9.2. Implement and run the code from the previous section (non-displacing multi-threading).

Exercise 9.3. Implement the transfer function in Lua. If a think about calling resume-yeild the same as calling function and return from it, then this function will be like a goto : it interrupts the current coroutine and resumes any other th coroutine passed as an argument.
(*Hint* : use an analog of the dispatch procedure to control handling your coroutines. Then transfer will transfer control to the dispatcher, informing about which next com- the program needs to be started, and the dispatcher will call re- sume).

Chapter 10
Completed examples

To conclude this introduction to the language, we show three simple, but complete programs. The first program solves the problem of eight kings wah. The second program prints the most common words in text. The last example is the implementation of the Markov chain described by Naya Kernighan and Pike in their book " *The Practice of Programming* " (Addison-Wesley, 1999).

10.1. The Eight Queens Problem

Our first example is a very simple program that solves *the problem of eight queens* : you need to arrange the eight queens on chessboard so that none of the queens are under attack.
The first step in solving this problem should be noted that each Each solution must have exactly one queen in each line. Thus, we can represent the solution as an array of eight numbers, one for each line; each number tells us in which column is the queen in the corresponding row. On-

example, the array {3,7,2,1,8,6,5,4} means one queen
is in row 1 in column 3, another is in row 2 in column 7 and
etc. (Note that this is not a valid solution
eat; for example, the queen in row 3 in column 2 attacks the queen in
line 4 in column 1). Also note that any solution
should be a permutation of numbers from 1 to 8, as the solution should
contain one queen in each column.
The complete program is shown in Listing 10.1. The first function is -
it is isplaceok , which checks that the given position on the board is not
falls under the battle of previously placed queens. Remembering that he cannot
to be two queens on the same line, this function checks that there are no
two queens on one column or one diagonal with a given
position.

Listing 10.1. Program for eight queens

```
local N = 8 - board size
- checks that position (n, c) is not under attack
local function isplaceok (a, n, c)
for i = 1, n - 1 do –- for each previously placed queen
if (a [i] == c) or –- the same column?
(a [i] - i == c - n) or –- the same diagonal?
(a [i] + i == c + n) then -– the same diagonal?
return false -– position under attack
end
end
return true -– not under attack
end
- print the board
local function printsolution (a)
for i = 1, N do
for j = 1, N do
io.write (a [i] == j and “X” or “-”, ““)
end
io.write (“\ n”)
end
io.write (“\ n”)
end
- add to the board 'a' all queens from 'n' to 'N'
local function addqueen (a, n)
if n> N then -– have all the queens been placed?
printsolution (a)
else - try to place the nth queen
for c = 1, N do
if isplaceok (a, n, c) then
a [n] = c –- put the n-th queen in column 'c'
addqueen (a, n + 1)
```

```
end
end
end
end
-- run the program
addqueen ({}, 1)
```

Next we have the printsolution function , which prints the check
mat board. She just goes around the board, typing 'X' in places with
queen and '-' elsewhere. Each result looks
dit like below:

```
The most common words
X - - - - - - -
- - - - X - - -
- - - - - - - X
- - - - - X - -
- - X - - - - -
- - - - - - X -
- X - - - - - -
- - - X - - - -
```

The last function addqueen is the heart of the program. Sleep-
then it checks if the solution is complete, and if so, then
prints this solution. Otherwise, it iterates over all the pillars.
tsy; for each unallocated column, the program puts there a short
left and recursively tries to accommodate the remaining queens.

10.2. Most often occurring words

Our next example is a simple program that reads
text and prints the most frequent words from that text.
The main data structure of this program is just a table
tsa, which matches each word with its frequency. Using
With this data structure, the program has three main tasks:

• Read the text by counting the number of occurrences of each word.
• Sort the list of words in descending order of frequency of meeting-
the bridges of every word.
• Print the first *n* items from the sorted list.

ka.

To read the text, we can use the allwords iterator , which
which we discussed in Section 7.1. For every word that we
read, we increment the corresponding counter:

```
local counter = {}
```

```
for w in allwords do
counter [w] = (counter [w] or 0) + 1
end
```

The next task is to sort the word list. However, how
an attentive reader might have noticed we don't have a word list! but
it is easy to create using words that are keys in the table
face counter:

```
local words = {}
for w in pairs (counter) do
words [#words + 1] = w
end
```

Listing 10.2. Program for printing the most common words

```
local function allwords ()
local auxwords = function ()
for line in io.lines () do
for word in string.gmatch (line, "% w +") do
coroutine.yield (word)
end
end
end
return coroutine.wrap (auxwords)
end
local counter = {}
for w in allwords () do
counter [w] = (counter [w] or 0) + 1
end
local words = {}
for w in pairs (counter) do
words [#words + 1] = w
end
table.sort (words, function (w1, w2)
return counter [w1]> counter [w2] or
counter [w1] == counter [w2] and w1 <w2
end)
for i = 1, (tonumber (arg [1]) or 10) do
print (words [i], counter [words [i]])
end
```

Now that we have a list, we can sort it with
using the table.sort function we discussed in Chapter 6:

```
table.sort (words, function (w1, w2)
return counter [w1]> counter [w2] or
counter [w1] == counter [w2] and w1 <w2
end)
```

The complete program is shown in Listing 10.2. Pay attention-
the use of coroutines in the auxwords iterator . In the last
in the loop that prints the result, the program considers that its first
the argument is the number of words to print and uses

value 10 if no arguments were passed.

10.3. Markov chain

Our final example is a *Markov chain* implementation . Program
generates pseudo-random text based on which words
can follow a sequence of *n* previous words in the text
st. For this implementation, we will assume that *n* is 2.
The first part reads the body text and builds a table that
for every two words gives a list of all the words that may be behind them
follow in the main text. After building the table, the program
uses it to construct a random text, where each word
follows the previous two with the same probability as in the base
in your text. As a result, we get text that is random, but not
absolutely. For example, applying it to the English text of this book,
we get texts like “Constructors can also traverse a table con-
structor, then the parentheses in the following line does the whole file in
a field n to store the contents of each function, but to show its only argu-
ment. If you want to find the maximum element in an array can return
both the maximum value and continues showing the prompt and running
the code. The following words are reserved and cannot be used to convert
between degrees and radians ".
We will encode each prefix by connecting two words when
using a space:

```
function prefix (w1, w2)
return w1 .. “ “ .. w2
end
```

We will use the string NOWORD (“\ n”) to initialize
prefix words and end-of-text symbols. For example, for text
"The more we try the more we do" the table of the following words will be
look like below:

```
{[“\ N \ n”] = {“the”},
[“\ N the”] = {“more”},
[“The more”] = {“we”, “we”},
[“More we”] = {“try”, “do”},
[“We try”] = {“the”},
[“Try the”] = {“more”},
[“We do”] = {“\ n”},
```

}

The program stores its table in the statetab variable . For to insert a new word into the table, we will use the following function:

```
function insert (index, value)
local list = statetab [index]
if list == nil then
statetab [index] = {value}
else
list [#list + 1] = value
end
end
```

It first checks that the given prefix already has a list; if not, it creates a new list with the passed word. Otherwise she inserts the passed word at the end of the existing list. To build the statetab we will use two variables w1 and w2 containing the last two words read. For each new word read, we add it to the list, associated with w1-w2 , and then update the values for w1 and w2 . After building the table, the program begins to build the text, with consisting of MAXGEN words. To begin with, she sets the values of the changes nym w1 and w2 . Then, for each prefix, she randomly chooses next word from the list of valid words, prints that word and updates the values of w1 and w2 . Listings 10.3 and 10.4 contain the complete program. Unlike our previous example with the most common words, here we use the implementation allwords based on closures.

Listing 10.3. Additional program definitions with Markov chain

```
function allwords ()
local line = io.read () -– current line
local pos = 1
-– current position in the line
return function () -– iterating function
while line do
- repeat until there are lines left
local s, e = string.find (line, “% w +”, pos)
if s then
- found the word?
pos = e + 1 - update position
return string.sub (line, s, e) - return word
else
line = io.read () - the word was not found; let's try the trail. string
pos = 1
- start at the beginning of the line
```

```
end
end
return nil
–- no more lines, end of traversal
end
end
function prefix (w1, w2)
return w1 .. “ “ .. w2
end
local statetab = {}
function insert (index, value)
local list = statetab [index]
if list == nil then
statetab [index] = {value}
else
list [#list + 1] = value
end
end
```

Listing 10.4. Markov chain program

```
local N = 2
local MAXGEN = 10000
local NOWORD = “\ n”
- build a table
local w1, w2 = NOWORD, NOWORD
for w in allwords () do
insert (prefix (w1, w2), w)
w1 = w2; w2 = w;
end
insert (prefix (w1, w2), NOWORD)
- generate text
w1 = NOWORD; w2 = NOWORD - initialize
for i = 1, MAXGEN do
local list = statetab [prefix (w1, w2)]
- choose a random word from the list
local r = math.random (#list)
local nextword = list [r]
if nextword == NOWORD then return end
io.write (nextword, “ “)
w1 = w2; w2 = nextword
end
```

Exercises

Exercise 10.1. Change the program with eight queens, so that it stops after printing the first decision.

Exercise 10.2. An alternative implementation of the problem of eight queens can be the construction of all permutations of numbers from

1 through 8 and check which ones are valid. Change the program
mu for using this approach. How fast is
the effect of the new program in comparison with the old one?
(*Hint* : Compare the total number of permutations with the number of times
when the original program calls the isplaceok function .)
Exercise 10.3. When we use the program to determine
the most common words, then usually the most common
the most common words are short, uninteresting words
like articles and prepositions. Modify the program so that
she skipped words of less than four letters.
Exercise 10.4. Generalize the Markov chain algorithm so that-
could use any size as length
prefix.

Part II
T TABLES
AND OBJECTS

CHAPTER 11
Data structures

Lua tables are not just a data structure, they are basic and
single data structure. All structures that offer
other languages - arrays, records, lists, queues, sets - can

be represented in Lua using tables. Moreover, the tables in
Lua effectively implements all of these structures.
In traditional languages such as C and Pascal, we implement more
most data structures using arrays and lists (where the list
ki = records + pointers). Although we can implement arrays and
lists with tables in Lua (and sometimes we do), tables are
much more powerful than arrays and lists; many algorithms with
using tables become almost trivial. For instance,
we rarely use Lua lookups as tables provide
direct access to values of various types.
It takes time to understand how to effectively use tab-
faces in Lua. In this chapter, I will show you how to implement typical
structures.
rounds of data using tables, and I will give examples of their use
niya. We won't start with arrays and lists because we need them.
us for other structures, but since most programmers
already familiar with them. We have already seen the basis of this material in the pre-
in previous chapters, but I will also repeat it here.

11.1. Arrays

We are implementing arrays in Lua by simply indexing tables with integers
numbers. So arrays are not fixed size
and grow as needed. Usually, when initializing an array, we
implicitly set its size. For example, after doing the following
code, any attempt to access a field outside the range of 1-1000 will return
nil instead of 0:

```
a = {} - new array
for i = 1, 1000 do
a [i] = 0
end
```

The length operator ('#') uses this to define the length of the mass-
Siwa:

```
print (#a) -> 1000
```

You can start an array from zero or any other value:

```
- create an array with indices from -5 to 5
a = {}
for i = -5, 5 do
a [i] = 0
```

```
end
```
However, it is common in Lua to start arrays at index 1. Bib-
the Lua libraries follow this convention; as well as the operator length
us. If your arrays don't start at 1, then you won't be able to use
call these capabilities of the language.
We can use the constructor to create and initialize-
array of one expression:
```
squares = {1, 4, 9, 16, 25, 36, 49, 64, 81}
```
Such constructors can be so large that
how much is needed (at least up to several million
cops).

11.2. Matrices and multidimensional arrays

There are two main ways of representing matrices in Lua. First -
is to use an array of arrays, i.e. a table, each element
which is another table. For example, you can create a matrix
zu of zeros of size M by N using the following code:
```
mt = {}
-- create matrix
for i = 1, N do
mt [i] = {} -- create string
for j = 1, M do
mt [i] [j] = 0
end
end
```

Since tables are objects in Lua, to create mat-
You must explicitly create each row of the script. On the one hand, it is
is more cumbersome than simply declaring a matrix, as is done in
languages C and Pascal. On the other hand, it gives more flexibility. For
instance,
you can create a triangular matrix by changing the `for` loop `j = 1, M
do ... end` in the previous code snippet `for j = 1, i do ... end` .
With this code, the triangular matrix will only use the posi-
memory fault over the original example.
The second way to represent matrices in Lua is to combine
converting two indices into one. If both indices are integers
lami, then you can simply multiply the first by the corresponding
constant and add a second index. With this approach the following

the code will create our matrix of zeros of size M by N :

```
mt = {} -– create matrix
for i = 1, N do
for j = 1, M do
mt [(i - 1) * M + j] = 0
end
end
```

If the indices are strings, then you can create one in-
dex by simply concatenating those lines with some character in between.
For example, you can create a matrix m with string indices-
mi s and t using the following code m [s .. ":" .. t] , provided that
that both s and t do not contain colons; otherwise pairs like
("A:", "b") and ("a", ": b") will both give the same index "a :: b" .
When in doubt, you can use the control sim
ox like ' \ 0' for separating indices.
Quite often, applications use a *sparse matrix* ,
that is, a matrix where most of the elements are either 0 or ***nil*** . On-
example, you can represent a graph using its connectivity matrix-
value in which the value at position m, n is equal to x , if between nodes
m and n is a join at a cost x. When these nodes are not connected, then the value
nil at position m, n is ***nil*** . In order to represent a graph with ten
with thousands of nodes, where each node has about five neighbors, you
a matrix with one hundred million possible elements is needed, but only
fifty thousand of them will be non- ***nil*** (five non-zero columns
for each row corresponding to five neighbors). Many books
on data structures discuss in detail how you can implement
similar sparse matrices, without wasting 400 MB of memory on them, but
you rarely need such tricks when programming
in Lua. With our first view (table of tables) you will
ten thousand tables are needed, each of which contains about
five elements, that is, a total of about fifty thousand values. When
the second view we have one table with fifty thousand-
elements. Whichever presentation you use, you
only memory is needed for non- ***nil*** elements .
When working with sparse matrices, we cannot use
length operator due to holes (***nil*** values) between elements. One-
but this is not a big loss; even if we could use it, then
it wouldn't be worth it. For most operations, it was extremely
it is inefficient to iterate over all these empty elements. Instead, we
we can use pairs to traverse only elements other than

from ***nil*** . For example, in order to multiply a string by a constant, we we can use the following code:

```
function mult (a, rowindex, k)
local row = a [rowindex]
for i, v in pairs (row) do
row [i] = v * k
end
end
```

Note, however, that the keys do not have any specific divided order in the table, so iterating with pairs does not guarantee that we will visit all columns in ascending order. For some tasks (for example, our previous example) this is no problem. For other purposes, you can use the excellent views, such as linked lists.

11.3. Linked Lists

Since tables are dynamic entities, re-
It's pretty easy to lick linked lists in Lua. Each node represented by a table, and the links are just fields of the table, which contain links to other tables. For example, let's re-we lick the simplest list, where each node contains two fields, next and value . The root of the list is an ordinary variable:

```
list = nil
```

To insert an element with the value v at the beginning of the list, we do:

```
list = {next = list, value = v}
```

To traverse the list, we can write:

```
local l = list
while l do
<visit l.value>
l = l.next
end
```

Other list options, such as bidirectional or circular lists are also easy to implement. However, such structures you will rarely need Lua as there is usually an easier one a way of presenting your data without using related lists. For example, we can represent the stack as (unlimited ny) array.

11.4. Queues and doubles queues

The simplest way to implement queues in Lua is to use
The insert and remove functions from the table library . These functions
insert and remove elements from an arbitrary position in the array,
moving the rest of the array elements. However, such changes
Scales can be expensive for large structures. More efficient
nth implementation uses two indices, one for the first element and
one for the latter:

```
function ListNew ()
return {first = 0, last = -1}
end
```

In order not to pollute the global namespace, we
define all operations for working with a list within a table, which
We'll call it List (this way we'll create a *module*). Then we
we can rewrite our last example as follows:

```
List = {}
function List.new ()
return {first = 0, last = -1}
end
```

Now we can insert and remove elements from either end beyond
constant time:

```
function List.pushfirst (list, value)
local first = list.first - 1
list.first = first
list [first] = value
end
function List.pushlast (list, value)
local last = list.last + 1
list.last = last
list [last] = value
end
function List.popfirst (list)
local first = list.first
if first> list.last then error (“list is empty”) end
local value = list [first]
list [first] = nil - let the garbage collector remove it
list.first = first + 1
return value
end
function List.poplast (list)
local last = list.last
if list.first> last then error (“list is empty”) end
local value = list [last]
list [last] = nil - let the garbage collector remove it
```

```
list.last = last - 1
return value
end
```

If you use this structure in the classic way
bong, calling only pushlast and popfirst , then both first and last will be
grow constantly. However, since we represent arrays in Lua with
help tables, you can easily index them from 1 to 20 or from
16,777,216 to 16,777,236. Since Lua uses double
precision to represent numbers, your program can run
for two hundred years, making a million insertions per second,
before an overflow problem occurs.

11.5. Sets and sets

Suppose you want to iterate over all ids used by
in the program; somehow you need to filter
reserved words. Some C programmers can
try to use to represent the set of reserved
given words an array of strings and then to check if
the given word is reserved, search in this
sive. You can even use the binary to speed up searches.
tree to represent the set.
In Lua, an efficient and easy way to represent sets
will use the elements as *indexes* on the table. Then instead of
to find if the table contains a given word, you can simply ask
try to index the table with this word and see if
whether the resulting result is ***nil*** . For example, we can use
the following code:

```
reserved = {
[“While”] = true, [“end”] = true,
[“Function”] = true, [“local”] = true,
}
for w in allwords () do
if not reserved [w] then
<do something with 'w'>
- 'w' is an unreserved word
end
end
```

(Since these words are reserved in Lua, we cannot use
use them as identifiers; for example, we cannot

write while = true . Instead, we write ["while"] = true .)
You can also use clearer initialization when
using an additional function that builds the set:

```
function Set (list)
local set = {}
for _, l in ipairs (list) do set [l] = true end
return set
end
reserved = Set {"while", "end", "function", "local",}
```

Sets, also called *multisets* , differ from
ordinary sets in that each element may not occur
how many times. The simple representation of sets in Lua is similar to the pre-previous view for sets, but with each key associated
corresponding counter. In order to insert an element, we
Liching his counter:

```
function insert (bag, element)
bag [element] = (bag [element] or 0) + 1
end
```

To remove an element, we decrement its counter:

```
function remove (bag, element)
local count = bag [element]
bag [element] = (count and count> 1) and count - 1 or nil
end
```

We store the counter only if it already exists and is not equal
zero.

11.6. Line buffers

Suppose you are working with text and reading the file line by line. Then your code might look like this:

```
local buff = ""
for line in io.lines () do
buff = buff .. line .. "\ n"
end
```

Despite its harmless appearance, this code can hit hard
for performance for large files: for example, reading a file in
1 MB takes 1.5 minutes on my old computer [1] .
Why is this so? To understand what is happening, imagine that
we are inside a loop; each line is 20 bytes, and
we have already read 2500 lines, so buff is a 50 KB line. Kog-yes Lua connects buff..line .. "\ n" ; it allocates a new line in
50,020 bytes and copies 50,000 bytes from buff to this newline. Ta-

Thus, for each new line, Lua moves in memory when
approximately 50 Kb, and this size is only growing. More precisely this algorithm
has quadratic complexity. After reading 100 new lines
(2 KB total) Lua has already moved over 2 MB of memory. When Lua is
reads 350 KB, more than 50 GB will already be moved in memory
(this problem is not unique to Lua: other languages where strings
immutable, also face a similar problem, the most
Java is a well-known example of such a language).
Before we continue, it should be noted that, despite
all that said, this is not a typical problem. For small lines
the above loop works fine. To read the entire file
Lua provides io.read (“* a”) , this call reads the entire
file. However, sometimes we run into this problem. To fight
With a similar problem Java uses the StringBuffer structure .
In Lua, we can use a table as a string buffer.
The key to this approach is the table.concat function , which
returns the result of concatenating all strings from the given list.
With concat we can rewrite our previous code to the following
in the following way:

```
local t = {}
for line in io.lines () do
t [#t + 1] = line .. "\ n"
1
"My Old Computer" is a 3 GHz single-core 32-bit Pentium. Everything is fast
The performance for this book was measured on this computer.

end
local s = table.concat (t)
```

This algorithm takes less than 0.5 seconds to read the same self-
th file, which took almost a minute with the previously used
code. (Regardless, to read the entire file, it is best to use
io.read with the “* a” option .)
We can do even better. The concat function takes as input
second optional argument, which is a delimiter,
which will be inserted between the lines. Using this separa-
torus, we can get rid of the need to insert every time
character '\ n' :

```
local t = {}
for line in io.lines () do
t [#t + 1] = line
end
```

```
s = table.concat (t, "\ n") .. "\ n"
```
The concat function inserts a separator between lines, but we
you still need to add one last '\ n' character . This last
The new concatenation operation copies the resulting string, which can
require a significant amount of time. There is no way to force
concat insert an extra delimiter, but we can easily
achieve this by simply adding an empty line to t :
```
t [#t + 1] = ""
s = table.concat (t, "\ n")
```
An extra '\ n' character that concat will add before the post
the ice line is what we need.

11.7. Graphs

Like any sane language, Lua offers various implementations
for graphs, each of which is better suited for its own type of al-
burning. Here we will look at a simple object-oriented
a new implementation in which we will represent the nodes as objects
(more precisely, tables, of course) and arcs as links between nodes.
We will represent each node as a table with two fields:
name , which is the name of the node, and adj , which is the set of nodes.
fishing connected to the data. Since we will be reading the graph from the text
st file, we need a way to find the node by its name. For
for this we will use an additional table. Function na-
me2node , having received the name of the node, will return this node:
```
local function name2node (graph, name)
local node = graph [name]
if not node then
- there is no node yet, create a new one
node = {name = name, adj = {}}
graph [name] = node
end
return node
end
```
Listing 11.1 contains a function that will build a graph. It
reads a file where each line contains the names of two nodes, denoting
that there is an arc leading from the first node to the second. For each line
it uses the string.match function to split the string
ku on two names, then finds the corresponding nodes (creating them
if necessary) and connects them.

Listing 11.1. Reading a graph from a file

```
function readgraph ()
local graph = {}
for line in io.lines () do
- split the line into two names
local namefrom, nameto = string.match (line, "(% S +)% s + (% S +)")
- find matching nodes
local from = name2node (graph, namefrom)
local to = name2node (graph, nameto)
- add 'to' to the link list of node 'from'
from.adj [to] = true
end
return graph
end
```

Listing 11.2 illustrates an algorithm using similar graphs. The findpath function searches for a path between two nodes using depth-first traversal. Its first parameter is the current node; second forgives the desired node; the third parameter stores the path from the beginning to the current mu node; the last parameter is the set of all already visited nodes (to avoid loops). Notice how the algorithm works directly with nodes, avoiding the use of their names. Onexample, visited is a set of nodes, not node names. Similarly path is a list of nodes.

Listing 11.2. Finding a path between two nodes

```
function findpath (curr, to, path, visited)
path = path or {}
visited = visited or {}
if visited [curr] then -– has the node already been visited?
return nil
- there is no way
end
visited [curr] = true
-– mark the node as visited
path [#path + 1] = curr -– add to the path
if curr == to then
-- target?
return path
end
- try all neighboring nodes
for node in pairs (curr.adj) do
local p = findpath (node, to, path, visited)
if p then return p end
end
path [#path] = nil
–- remove a node from the path
end
```

To test this code, we'll add a function that prints
path, and additional code to make it work:

```
function printpath (path)
for i = 1, #path do
print (path [i] .name)
end
end
g = readgraph ()
a = name2node (g, “a”)
b = name2node (g, “b”)
p = findpath (a, b)
if p then printpath (p) end
```

Exercises

Exercise 11.1. Modify the implementation of the queue so that both
the index would be zero if the queue is empty.

Exercise 11.2. Repeat exercise 10.3, only instead of
in order to use length as a criterion for discarding
word, now the program should read from the special
th file list of words to skip.

Exercise 11.3. Modify the graph structure so that it matches
kept a label for each arc. Each arc must also
be represented using an object with two fields: met-
coy and knots to which she points. Instead of many
neighboring nodes each node must contain a list of arcs
going from this node.
Modify the readgraph function so that it is from each line
ki file read two node names and a label (assuming that the label
this number).

Exercise 11.4. Use the graph representation from the previous
the next exercise, where the label of each arc is
the distance between the nodes it connects. Write a function
tion that finds the shortest path between two nodes.
(*Hint* : Use Dijkstra's algorithm.)

CHAPTER 12
Data files and persistence

When working with data files it is usually much easier to write
files than to read them. When we write to a file, we are completely
we roll everything that happens. On the other hand, when we read
from the file, we don't know what to expect. Besides all data types,
which the correct data file can contain, the program
ma should also handle bad files intelligently. therefore
writing correctly working procedures for reading data
always difficult.
In this chapter, we will see how you can use Lua to do something
would eliminate all the code for reading data from our programs, just
writing data in a suitable format.

12.1. Data files

Table constructors provide an interesting alternative to the form
data mats. With a little extra work when
writing data reading becomes trivial. The approach is
is to write our data file as a Lua program that
Paradise, when executed, creates the necessary data.
As usual, for the sake of clarity, let's look at the
measures. If our data file is in a specific format,
for example CSV or XML, our choice is extremely small. However, if we
want to create a file for our own use, then we in
we can use Lua constructors as our format. IN
In this format, we represent each entry as a Lua constructor.
Instead of writing to our file something like

Donald E. Knuth, Literate Programming, CSLI, 1992
Jon Bentley, More Programming Pearls, Addison-Wesley, 1990

we're writing:

```
Data files
Entry {“Donald E. Knuth”,
"Literate Programming",
“CSLI”,
1992}
```

```
Entry {“Jon Bentley”,
“More Programming Pearls”,
"Addison-Wesley",
1990}
```

Let's remember? that `Entry { code }` is the same as `Entry ({ code })` , that is, a call to the `Entry` function with the table as the only one argument. Therefore, the above piece of data is by itself actually a Lua program. In order to read such a file, we you just need to execute it with a properly defined the `Entry` function . For example, the following program counts the number entries in the file:

```
local count = 0
function Entry () count = count + 1 end
dofile (“data”)
print (“number of records:“ .. count)
```

The following program builds a set of all author names, find data in the file, and prints them (not necessarily in the same order, in which they met in the file):

```
local authors = {} -– many authors
function Entry (b) authors [b [1]] = true end
dofile (“data”)
for name in pairs (authors) do print (name) end
```

Please note the approach used in these snippets code: `Entry` function acts as a callback function (callback), which is called at runtime `dofile` for each doy record in the file.

When we don't care about file size, we can use our views use name-value pairs [1] :

```
Entry {
author = “Donald E. Knuth”,
title = “Literate Programming”,
publisher = “CSLI”,
year = 1992
}
Entry {
author = “Jon Bentley”,
```

[1] If this format reminds you of BibTeX, then it's no coincidence. BibTeX format was one of the sources that defined the kind of constructors in Lua.

```
title = “More Programming Pearls”,
year = 1990,
publisher = “Addison-Wesley”,
}
```

This format is what we call a *self - describing format*

data, since each piece of data contains a short description
vanishing of its meaning. Self-describing data is more readable (like
at least people) than CSV or other compact format; their easy
to edit if necessary; and they allow us to contribute
small changes to the base format without the need to change
data files. For example, if we add a new field, then we
you only need to change the reading program by providing a value
by default when no field is specified.
Using the name-value format, our program for composing
The author list becomes as shown below:

```
local authors = {} -– set for author names
function Entry (b) authors [b.author] = true end
dofile (“data”)
for name in pairs (authors) do print (name) end
```

Now the order of the fields is not important. Even if some records
there is no author, then we only need to change the Entry function :

```
function Entry (b)
if b.author then authors [b.author] = true end
end
```

Lua is not only fast to execute, but also fast to compile.
For example, the above program for making a list
authors processes 1 MB of data in one tenth of a second [2] . And this
not by chance. Data description was one of the main applications
Lua since its inception, and we pay a lot of attention to making it
the compiler was fast for large programs.

12.2. Serialization

Often we need to serialize some data, that is, translate
data into a stream of bytes or characters that we can write
to a file or send over the network. We can represent serialized-
data as Lua code in such a way that when doing this
code it restores the saved values for the executing
his programs.
Usually, if we want to restore the value of the global re-
variable, then our block of code will be something like varname = *exp* , where *exp* is
this is the Lua code to get the value. With varname everything is simple, so
Let's see how to write code that creates a value.

For a numeric value, the task is simple:

```
function serialize (o)
if type (o) == “number” then
io.write (o)
else <other cases>
end
end
```

When writing a number in decimal form, there is a risk of losing precision.
In Lua 5.2, you can use hexadecimal format, in order to
to avoid a similar problem:

```
if type (o) == “number” then
io.write (string.format (“% a”, o))
```

When using this format (“% a”) the read value
will consist of exactly the same bits as the original one.
For a string, a naive approach would be something like the following:

```
if type (o) == “string” then
io.write (“‘”, o, “‘”)
```

However, if the string contains special characters (such as
tricks or '\ n'), then the resulting code will no longer be a program for
Lua.
You might think that this problem can be solved by changing
quote type:

```
if type (o) == “string” then
io.write (“[[“, o, “]]”)
```

However, be careful. If you try to keep something
like “]] .. os.execute ('rm *') .. [[” (for example, by passing this
string as an address), then the resulting block of code will be:

```
varname = [[]] .. os.execute ('rm *') .. [[]]
```

As a result, you will receive an unpleasant surprise when trying to purchase
honor is such "data".
The simplest way to write a string is safe to use
the option "% q" from the string.format function . She surrounds the string
double quotes and safely represents double
quotes and some other characters inside the string:

```
a = 'a “problematic” \\ string'
print (string.format (“% q”, a)) -> “a \” problematic \ ”\\ string”
```

Using this capability, our serialize function can
look like this:

```
function serialize (o)
if type (o) == “number” then
io.write (o)
elseif type (o) == “string” then
io.write (string.format (“% q”, o))
else <other cases>
end
end
```

Since version 5.1 Lua offers a different way of writing strings
in a safe way, using the notation [= [...] =] for long
lines. However, this recording method is mainly intended for
user-written code when we in no way want
change the character string. It is easier in auto-generated code
use “% q” from string.format .
If you nevertheless want to use a similar notation for
automatically generated code, then you need to pay attention
for some details. The first is that you need to pick up
the correct number of equal signs. A good option is
The number is greater than that found in the original string. Insofar as
strings containing a large number of equal signs are not

are rare (for example, comments separating blocks of code),
then we can restrict ourselves to considering sequences of
equality kovs between square brackets; others
sequences cannot result in an erroneous end marker
strings. The second detail is that Lua always ignores sym-
ox '\ n' at the beginning of a long line; the simplest way to deal with
this is the addition of the '\ n' character , which will be discarded.

Listing 12.1. Outputting an arbitrary string of characters

```
function quote (s)
- find the maximum length of a sequence of equal signs
local n = -1
for w in string.gmatch (s, “] = *]”) do
n = math.max (n, #w - 2) - -2 to remove ']'
end
- create a string with 'n' + 1 equal sign
local eq = string.rep (“=”, n + 1)
- build a summary line
return string.format (“[% s [\ n% s]% s]“, eq, s, eq)
end
```

The quote function in Listing 12.1 is the result of our
for previous remarks. It receives an arbitrary string as input and
returns the formatted string as a long string. Call
string.gmatch creates an iterator to iterate over all sequential
of the form '] = *]' (that is, the closing square bracket, after which
followed by zero or more equal signs, followed by
there is another closing square bracket) on line 3 . For each
occurrences are updated with the value n equal to the maximum number
already met equal signs. After the loop, we use the function
ktsiyu string.rep , to repeat the sign of equality n + 1 times,
i.e. one more than the maximum number encountered
in line. Finally, the function string.format concludes s between PA-
square brackets with the appropriate number of equal signs and
adds extra spaces around the line and the '\ n' character in
the beginning of the line.

Saving tables without loops

Our next (and more challenging) challenge is to preserve
tables. There are several ways to save them in accordance with

with what restrictions we impose on the structure of the table
tsy. There is no one algorithm that fits all cases.
Simple tables not only require simpler algorithms, but also
the resulting files can be visually pleasing.

Listing 12.2. Serializing tables without loops

```
function serialize (o)
if type (o) == “number” then
io.write (o)
elseif type (o) == “string” then
io.write (string.format (“% q”, o))
elseif type (o) == “table” then
io.write (“{\ n”)
for k, v in pairs (o) do
io.write (““, k, “=“)
serialize (v)
io.write (“, \ n”)
end
io.write (“} \ n”)
else
error (“cannot serialize a“ .. type (o))
end
end
```

Our next attempt is shown in Listing 12.2. Despite
for its simplicity, this function does a pretty decent job.
It even handles nested tables (i.e. tables inside
other tables) as long as the table structure is a tree
(that is, there are no shared sub-tables and loops). A little visual
an improvement would be to add spaces to indent nested
tables (see Exercise 12.1).
The previous function assumes that all keys in the table are
are valid identifiers. If the table contains numeric
keys or strings that are not identifiers in Lua, then
we have a problem. A simple way to resolve it is to use
Writing the following code to write each key:

```
io.write (“[“); serialize (k); io.write (“] =“)
```

With this improvement, we have increased the reliability of our function.
due to the visual clarity of the resulting file. Consider
next call:

```
serialize {a = 12, b = 'Lua', key = 'another “one”'}
```

The result of this call when using the first version of the function
tion serialize will the following code:

```
{
a = 12,
b = “Lua”,
key = “another \” one \ ””,
}
```

Compare with the result of using the second version:

```
{
[“A”] = 12,
[“B”] = “Lua”,
[“Key”] = “another \” one \ ””,
}
```

We can get both reliability and a beautiful view by checking in
in each case, whether square brackets are needed; again we will leave it
improved as exercise.

Saving tables with loops

For processing tables in the general case (that is, with loops and general subtables), we need a different approach. Constructors
cannot represent such tables, therefore we will not
use. We need names to represent loops, so
our next function will take the value as arguments
to save and name. Moreover, we must keep track of the names
already saved tables in order to reuse them when
we discover a cycle. To do this, we will use an additional
See the table. This table will use tables as
indexes and their names as stored values.

Listing 12.3. Saving tables with loops

```
function basicSerialize (o)
if type (o) == “number” then
return tostring (o)
else -– suppose it's a string
return string.format (“% q”, o)
end
end
function save (name, value, saved)
saved = saved or {} –- initial value
io.write (name, “=“)
if type (value) == “number” or type (value) == “string” then
io.write (basicSerialize (value), “\ n”)
elseif type (value) == “table” then
if saved [value] then -– is the value already saved?
io.write (saved [value], “\ n”) -– use its name
```

```
else
saved [value] = name -– save the name for next time
io.write (“{} \ n”) -– create a new table
for k, v in pairs (value) do — save its fields
k = basicSerialize (k)
local fname = string.format (“% s [% s]”, name, k)
save (fname, v, saved)
end
end
else
error (“cannot save a“ .. type (value))
end
end
```

The resulting code is shown in Listing 12.3. We stick with constraints that the tables we want to keep contain just numbers and strings as keys. BasicSerialize function serializes these base types. The next function, save , does em all the hard work. The parameter saved is a table that is monitors tables already saved. For example, if we build table as follows:

```
a = {x = 1, y = 2; {3,4,5}}
a [2] = a
- cycle
az = a [1]
- general subtable
```

then calling save (“a”, a) will save it like this:

```
a = {}
a [1] = {}
a [1] [1] = 3
a [1] [2] = 4
a [1] [3] = 5
a [2] = a
a [“y”] = 2
a [“x”] = 1
a [“z”] = a [1]
```

The order of these assignments can change, as it depends from traversing the table. Nevertheless, the algorithm guarantees that any the element required for building the table has already been defined .

If we want to store multiple values with common parts, then we can call the save function on the same table saved .

For example, consider the following two tables:

```
a = {{“one”, “two”}, 3}
b = {k = a [1]}
```

If we keep them independently, then the result will have no common

parts:

```
save ("a", a)
save ("b", b)
-> a = {}
-> a [1] = {}
-> a [1] [1] = "one"
-> a [1] [2] = "two"
-> a [2] = 3
-> b = {}
-> b ["k"] = {}
-> b ["k"] [1] = "one"
-> b ["k"] [2] = "two"
```

However, if we use the same `saved` table for both calls to `save`, then the resulting result will contain the common parts:

```
local t = {}
save ("a", a, t)
save ("b", b, t)
-> a = {}
-> a [1] = {}
-> a [1] [1] = "one"
-> a [1] [2] = "two"
-> a [2] = 3
-> b = {}
-> b ["k"] = a [1]
```

As usual, there are several other options in Lua. Among them, we can store the value without giving it a global name nor (for example, a block builds a local value and returns it), we we can process functions (by constructing additional the table associating each function with its name), etc. Lua gives you have strength; you build mechanisms.

Exercises

Exercise 12.1. Modify the code in Listing 12.2 to make it equated nested tables.
(*Hint* : add an extra function parameter `serialize` containing the alignment string.)

Exercise 12.2. Modify the code in Listing 12.2 so that it used the syntax `["key"] = value` as suggested in section 12.1.

Exercise 12.3. Modify the code of the previous exercise so that
to make it use the syntax `["key"] = value` only when
it's necessary.
Exercise 12.4. Modify the code of the previous exercise so that
so that it uses constructors whenever possible-
but. For example, he should present the table `{14,15,19}`
as `{14,15,19}` , not as `{[1] = 14, [2] = 15, [3] = 19}` .
(*Hint* : start by storing the values for keys 1, 2, ...,
until they are ***nil*** . Please note what is not needed
save them again when traversing the rest of the table.)
Exercise 12.5. The no-use approach
calling constructors when saving tables with loops,
too radical. You can save the table for more
nice way, using constructors in general and
then using assignments only to handle common
tables and loops.
Reimplement the `save` function using this
approach. Add to it everything that you have already implemented in the previous
exercises.

CHAPTER 13
Metatables and metamethods

Usually, for every value in Lua, there is a completely predictable
boron of operations. We can add numbers, connect strings, insert
insert key-value pairs into tables, etc. However, we cannot add
create tables, we cannot compare functions and we cannot call
string. Unless we're using metatables.
Metatables allow you to change the behavior of a value in case
when we are faced with an unexpected operation. For example, when
the power of metatables we can determine how Lua should compute
expression `a + b` , where `a` and `b` are tables. When Lua tries to fold
two tables, then it checks if at least one of them has metatab-
person and whether this metatable contains an `__add` field . If Lua finds
this field, then it calls the corresponding value - the so-called
my *metamethod* , which must be a function - to calculate

the amount.
Every value in Lua can have a metatable associated with it.
Tables and values of type userdata store individual values
for each instance; values of other types use
one general table for each type. Lua always creates new tables.
without metatables:

```
t = {}
print (getmetatable (t))
-> nil
```

We can use the setmetatable function to
set or change the metatable for any table:

```
t1 = {}
setmetatable (t, t1)
print (getmetatable (t) == t1)
-> true
```

Directly from Lua, we can set metatables
only for tables; to work with metatables of values of other types
so we have to use the C [1] code . We will see later in chapter 21,
that the string library sets up metatables
for strings. All other types do not have metatables by default:

```
print (getmetatable (“hi”)) -> table: 0x80772e0
print (getmetatable (“xuxu”)) -> table: 0x80772e0
print (getmetatable (10))
-> nil
print (getmetatable (print)) -> nil
```

Any table can be a metatable of any value; Group related tables can share a common metatable, which defines their general behavior; the table can be metatable- to herself so that she describes her own behavior nie.

13.1. Arithmetic metamethods

In this section, we will look at a simple example for defining
clarify how to use metatables. Let us use the table
ts for representing sets with functions for calculating the
connections, intersections, and so on, as shown in Listing 13.1. For
in order not to litter the global namespace, we will store
these functions are in the Set table .

Listing 13.1. Simple set implementation

```
Set = {}
- create a new set by taking values from a given list
function Set.new (l)
```

```
local set = {}
for _, v in ipairs (l) do set [v] = true end
return set
end
function Set.union (a, b)
local res = Set.new {}
for k in pairs (a) do res [k] = true end
for k in pairs (b) do res [k] = true end
return res
end
function Set.intersection (a, b)
local res = Set.new {}
for k in pairs (a) do
```
1
The main reason for this limitation is the desire to limit too much
frequent use of metatables. Experience with previous versions of Lua has shown that
such global changes often lead to unused code.
```
res [k] = b [k]
end
return res
end
- represent the set as a string
function Set.tostring (set)
local l = {} -– a list where all elements will be placed
for e in pairs (set) do
l [#l + 1] = e
end
return “{“ .. table.concat (l, “,“) .. “}”
end
- print set
function Set.print (s)
print (Set.tostring (s))
end
```
We will now use the addition operator ('+') to calculate
numbering the union of two sets. To do this, we will make it so that
all tables representing sets will have one common
metatable. This metatable will define how the tables should react
add to the addition operator. Our first step will be to create
a regular table that we will use as a metatable
for sets:
```
local mt = {} -– metatable for sets
```
The next step is to change the function that creates a lot of
the set Set.new . The new version of this function will have one additional
a string that sets mt for the generated tables
as a metatable:
```
function Set.new (l) -– 2nd version
local set = {}
```

```
setmetatable (set, mt)
for _, v in ipairs (l) do set [v] = true end
return set
end
```

After that, each set that we create with
Set.new will have the same metatable:

```
s1 = Set.new {10, 20, 30, 50}
s2 = Set.new {30, 1}
print (getmetatable (s1)) -> table: 00672B60
print (getmetatable (s2)) -> table: 00672B60
```

Finally, we'll add a metamethod to the metatable, the __add field , which
The swarm determines how the addition should be performed:

```
mt .__ add = Set.union
```

After that, whenever Lua tries to add two
union, it will call the Set.union function , passing both
rand as arguments.
With the metamethod, we can use the addition operator for
performing set union:

```
s3 = s1 + s2
Set.print (s3)
-> {1, 10, 20, 30, 50}
```

Similarly, we can define the multiplication operator for
completing the intersection of sets:

```
mt .__ mul = Set.intersection
Set.print ((s1 + s2) * s1) -> {10, 20, 30, 50}
```

For each arithmetic operator there is a corresponding
the name of the field in the metatable. Besides __add and __mul , there is also
__sub (for subtraction), __div (for division), __unm (for negation),
__mod (for taking the remainder from division) and __pow (for raising the
stump). We can also define a __concat field to specify the opera-
concatenation generator.
When we add two sets, the question is what meta-
to take the table, does not arise. However, we can write the expression,
which involves two values with different metatables, for example
measures as shown below:

```
s = Set.new {1,2,3}
s = s + 8
```

When looking for a metamethod, Lua takes the following steps: if y
the first value is a metatable with a __add field , then Lua uses
calls the corresponding value as a metamethod regardless
from the second value; otherwise, if the second value has metatables
tsu with the __add field , then Lua uses this value as a

there is a method; otherwise, an error occurs. In this way,
the last example will call Set.union , just like for
expressions 10 + s and "hello" + s .
Lua doesn't care about mixing types, but it is important
but for our application. For example, if we execute s = s + 8 , then
we get an error inside Set.union :
bad argument # 1 to 'pairs' (table expected, got number)
If we want to receive more accurate error messages, then
we must explicitly check the types of the operands before executing the opera-
walkie-talkies:

```
function Set.union (a, b)
if getmetatable (a) ~ = mt or getmetatable (b) ~ = mt then
error ("attempt to 'add' a set with a non-set value", 2)
end
<as before>
```

Remember that the second argument to the error function (2 in our case)
directs the error message to where the given operation was performed
called.

13.2. Comparison Methods

Metatables also allow you to make sense of operators compared
neniya using metamethods __eq (*equal*), __lt (*less than*)
and __le (*less than or equal to*). No special metamethods for three
remaining comparison operations: Lua translates a ~ = b to not (a == b) ,
a> b in b <a and a> = b in b <= a .
Prior to version 4.0, Lua translated all ordering operations into one,
translating a <= b to not (b <a) . However, such a translation is incorrect when
yes we are dealing with *partial ordering* , that is, when not all
the elements of our type are properly ordered. For instance,
floating point numbers are not fully ordered
on most computers because of the *NaN* (*Not a Number*) value .
In accordance with the IEEE 754 standard, NaN represents an undefined
specific values such as 0/0. According to the standard, any comparable
Nation including NaN must be false. It means that
NaN <= x is always false, but x <NaN is also false. It follows that
translating a <= b to not (b <a) is wrong in this case.
In our example with sets, we are dealing with a similar trial
lem. An obvious (and useful) value for <= for sets

is an occurrence of the set: a <= b means that a is a subset
property b . With this value, it is again possible that a <= b and b <a are false;
thus we need separate implementations for __le (less
or equal) and __lt (less than):

```
mt .__ le = function (a, b) -– occurrence of sets
for k in pairs (a) do
if not b [k] then return false end
end
return true
end
mt .__ lt = function (a, b)
return a <= b and not (b <= a)
end
```

Finally, we can define equality of sets in terms of embedding
sets:

```
mt .__ eq = function (a, b)
return a <= b and b <= a
end
```

After these definitions, we are ready to compare sets:

```
s1 = Set.new {2, 4}
s2 = Set.new {4, 10, 2}
print (s1 <= s2)
-> true
print (s1 <s2)
-> true
print (s1> = s1)
-> true
print (s1> s1)
-> false
print (s1 == s2 * s1) -> true
```

For types that have a complete ordering, we may not define
redistribute metamethod __le . If not, Lua uses __lt .
Comparison for equality also has some restrictions -
mi. If two objects have different base types or metamethods, then
comparison operation for equality will return ***false*** without even calling meta-
methods. Thus, the set will always be different from the number,
no matter what the metamethod returns.

13.3. Library metamethods

So far, we've seen metamethods defined in Lua itself.
The virtual machine itself checks if the values contain
Operation-defined metatables with corresponding metamethods.

However, since metatables are regular tables, then
anyone can use them. Therefore, libraries often define
have their own fields in metatables.
The tostring function is a typical example. As we have seen
earlier, tostring represents tables in a fairly simple way:

```
print ({})
-> table: 0x8062ac0
```

The print function always calls tostring to format
output. However, when formatting an arbitrary value
Nia tostring first checks whether the values metamethod
__tostring . If such a metamethod exists, then tostring calls it,
passing it an object as an argument. What will return this meta-
method, and will be the result of tostring .
In our set example, we have already defined a function for
representations of the set as strings. Therefore, we only need you-
put the __tostring field in the metatable:

```
mt .__ tostring = Set.tostring
```

After that, whenever we call print with a set as
with the same argument, print will call tostring , which in turn will
calls Set.tostring :

```
s1 = Set.new {10, 4, 5}
print (s1) -> {4, 5, 10}
```

The setmetatable and getmetatable functions also use meta-
field, in this case to protect the metatable. Suppose you
want to protect your sets so that users cannot
see nor modify their metatables. If you set the __metatable field
in a metatable, then getmetatable will return the value of that field, and calling
setmetatable will throw an error:

```
mt .__ metatable = “not your business”
s1 = Set.new {}
print (getmetatable (s1)) -> not your business
setmetatable (s1, {})
stdin: 1: cannot change protected metatable
```

In Lua 5.2, pairs and ipairs also have metatables, so
the table can change its workaround (or add a workaround
for non-table objects).

13.4. Metamets for access to the table

Metamets for arithmetic and comparison operations
define behavior for situations that would otherwise lead
to errors. They do not change normal behavior
language. But Lua also provides a way to change
the behavior of tables in two usual cases, reading and changing is
existing field in the table.

__Index metamethod

I said earlier that when we refer to a missing field
in the table, the result is ***nil*** . This is true, but this is not the whole truth.
In fact, such an appeal leads to the fact that the interpretation
The tator looks for the __index metamethod : if there is no such method, which usually
happens, then ***nil is*** returned ; otherwise the result is provided by the given metamethod.
The standard example here is inheritance. Let us
we want to create several tables describing windows. Each table
tsa must set various parameters of the window, such as position,
size, color scheme, etc. For all these parameters there is a value
default and therefore we want to build windows by setting only
those values that differ from the default values. Per-
your choice is a constructor that fills in the missing
fields. The second option is to arrange the windows in such a way
so that they *inherit* any missing field from the base proto-
type. First, we will declare a prototype and a constructor that will
creates new windows with a common metatable:

```
- create prototype with default values
prototype = {x = 0, y = 0, width = 100, height = 100}
mt = {} - create metatable
- declare a constructor function
function new (o)
setmetatable (o, mt)
return o
end
```

We will now define the __index metamethod :

```
mt .__ index = function (_, key)
return prototype [key]
end
```

After that, we will create a new window and access the missing
th field:

w = new {x = 10, y = 20}
print (w.width) -> 100
Lua specifies that w does not have the required field, but does have metatables.
with the __index field . Therefore, Lua calls this metamethod with an argument
ments of w (table) and "width" (missing field). Metamethode
accesses this field to the prototype and returns the resulting value
reading.
Using the __index metamethod for inheritance in Lua
so common that Lua provides a simplified version.
Despite the name of the *method* , the __index metamethod does not have to be
function: for example, it can be a table. When he is
function, then Lua calls it passing the table and missing-
key as arguments, as we have already seen. When this is a table, then
Lua simply accesses this table. Therefore, in our
in the previous example, we could simply define __index as follows-
in a way:
mt .__ index = prototype
Now when Lua searches for the __index metamethod , it will find
prototype value , which is a table. Accordingly, Lua
performs access to this table, that is, it performs an analog
prototype ["width"] . This appeal gives the required result.
Using a table as the __index metamethod makes it easy to
one and a quick way to implement the usual (not multiple) na-
followings. The function is a more expensive option, but
and provides more flexibility: we can implement
multiple inheritance, caching, and more. We are
we judge these forms of inheritance in chapter 16.
When we want to access a table without calling the metamethod
__index , then we use the rawget function . Calling rawget (t, i)
performs a direct access to the table t , that is,
scheduling without using metatables. Executing directly-
this call will not speed up your code (the cost of the function call will destroy
all that can be won), but sometimes it turns out to be necessary,
as we will see later.

__Newindex metamethod

The __newindex metamethod is analogous to the __index metamethod , but
only it works for writing values to a table. When you assign-

if you give the value to the missing field in the table, then the interpreter looks for the __newindex metamethod : if it exists, then the interpreter calls instead of doing the assignment. Like __index if metamethod is a table, then the interpreter performs the assignment for this table instead of the original one. Moreover, there is a function an operation that performs direct access, bypassing metamethods: rawset (t, k, v) writes the value v by key k to table t , not calling no metamethods.
Using the __index and __newindex Metamethods Together allows you to implement in Lua various rather powerful constructs tions such as read-only tables, tables with defaults and inheritance for object-oriented bathroom programming. In this chapter, we will see some of the their applications. Object-oriented programming a separate chapter is allocated.

Tables with default values

The default value for any field in a regular table is ***nil*** .
It's easy to change this behavior with metatables:

```
function setDefault (t, d)
local mt = {__index = function () return d end}
setmetatable (t, mt)
end
tab = {x = 10, y = 20}
print (tab.x, tab.z) -> 10 nil
setDefault (tab, 0)
print (tab.x, tab.z) -> 10 0
```

After calling setDefault, any call to the missing field in tab will call its __index metamethod , which will return zero (value d for this metamethod).
The setDeafult function creates a new closure and a new metatable. zu for each table that needs a default value. it can be costly if we have many tables that needs default values. The metatable has a default value NIJ d «sewn up" in her metamethod, so we can not use the same metatable for all tables. So that you can was to use the same metatable for tables with different default values, we can remember the default value in the table itself, using a special field for this. If a

not think about possible name conflicts, then we can use
call a key like "___" for our field:

```
local mt = {__index = function (t) return t .___ end}
function setDefault (t, d)
t .___ = d
setmetatable (t, mt)
end
```

Note that now we are creating the mt table only one
times, outside of the setDefault function .
If we want to guarantee the uniqueness of the key, then this is
but easy to provide. All we need is to create a new table
and use it as a key:

```
local key = {} - unique key
local mt = {__index = function (t) return t [key] end}
function setDefault (t, d)
t [key] = d
setmetatable (t, mt)
end
```

Another way to associate a default value with each
table is the use of a separate table, where the keys
are the tables themselves, and the values are the default values.
However, for the correct implementation of this approach, we need special
a special type of tables called *weak tables* ,
therefore, we will not use this approach here; we will return-
See this in chapter 17.
Another option is to remember metatables, whereby
we can reuse metatables corresponding to one
the same default value. However, this also requires
using weak tables, so we'll have to wait until
Chapter 17.

Tracking table access

Both __index and __newindex work only when in the table
no corresponding value. Therefore, the only way to
to keep track of all access to a table is to keep it empty. Thus,
zom, if we want to track all access to the table, then we need
create a special *proxy* table for the source table. She will-
det empty with appropriate __index and __newindex metamets
to track access to the table that will redirect
access to the original table. Let t be the original table, access

to which we want to track. Then we can use the following
blowing code:

```
t = {} - the source table was created somewhere
- create private access to it
local _t = t
- create a proxy
t = {}
- create a metatable
local mt = {
__index = function (t, k)
print ("* access to element" .. tostring (k))
return _t [k] - access to the source table
end,
__newindex = function (t, k, v)
print ("* update of element" .. tostring (k) ..
"To" .. tostring (v))
_t [k] = v - change the original table
end
}
setmetatable (t, mt)
```

This code keeps track of every access to t :

```
> t [2] = "hello"
* update of element 2 to hello
> print (t [2])
* access to element 2
hello
```

If we want to be able to traverse such a table, then we
you need to create a __pairs metamethod in the proxy table :

```
mt .__ pairs = function ()
return function (_, k)
return next (_t, k)
end
end
```

It is also possible to create something similar for __ipairs .
If we want to track access to multiple tables, then we
there is no need to create a separate metatable for each of them. Inmes-
then we can somehow link the proxy table with the original and
use one common metatable for all proxy tables. it
is similar to the task of linking a table with a default value,
which we looked at earlier. For example, you can store the outcome
a new table in a special field of the proxy table, using for this
special key. As a result, we end up with the following code:

```
local index = {} -– create a unique key
local mt = {- create metatable
__index = function (t, k)
```

```
print ("* access to element" .. tostring (k))
return t [index] [k] - access to the source table
end,
__newindex = function (t, k, v)
print ("* update of element" .. tostring (k) ..
"To" .. tostring (v))
t [index] [k] = v - change the source table
end,
__pairs = function (t)
return function (t, k)
return next (t [index], k)
end, t
end
}
function track (t)
local proxy = {}
proxy [index] = t
setmetatable (proxy, mt)
return proxy
end
```

Now, when we want to keep track of the table `t` , all we need is but, is to execute `t = track (t)` .

Read-only tables

It is easy to use the concept of proxy tables to create tables with pre-read-only mortar. All we need is to trigger an error every time we catch an attempt to change the table. For metame-Toda `__index` we can use the very original table instead functions, since we don't need to keep track of all reads from it; faster and more efficiently redirect such requests directly to the original table-face. This will require, however, a new metatable for each proxy-tables with an `__index` field pointing to the original table:

```
function readOnly (t)
local proxy = {}
local mt = {- create metatable
__index = t,
__newindex = function (t, k, v)
error ("attempt to update a read-only table", 2)
end
}
setmetatable (proxy, mt)
return proxy
end
```

As an example of a read-only table, we can

Let's create a table of names of days of the week:
days = readOnly {“Sunday”, “Monday”, “Tuesday”, “Wednesday”,
“Thursday”, “Friday”, “Saturday”}
print (days [1]) -> Sunday
days [2] = “Noday”
stdin: 1: attempt to update a read-only table

Exercises

Exercise 13.1. Define a __sub metamethod that returns
the difference between the two sets. (The set ab is the set
in all elements from a that are not contained in b .)
Exercise 13.2. Determine metamethod __len so that #s WHO-
rotates the number of elements in s .
Exercise 13.3. Complete the implementation of proxy tables in section
le 13.4 with the __ipairs metamethod .
Exercise 13.4. Another way to implement tables, access
read-only, is to use the function as
as the __index metamethod . This approach makes access to tab-
more expensive person, but the creation of such tables is cheaper,
as all read-only tables can have
one common metatable. Rewrite the readOnly function with
using this approach.
Exercises

Chapter 14
Environment

Lua stores all of its global variables in a regular table,
called the *global environment* . (More precisely, Lua
keeps its "global" variables in several environments, but
we will ignore this at first for simplicity.)
the advantage of this approach is that it simplifies the internal
Lua implementation as there is no need for special structure
data tour for storing global variables. Another advantage
the fact is that we can work with this table in the same way
as with any other table. To make things easier, Lua
stores the environment itself in the global variable _G . (Yes, _G._G is equal

but _G .) For example, the following code prints the names of all global variables defined in the global environment:

```
for n in pairs (_G) do print (n) end
```

In this chapter, we will see some useful methods for working with the environment.

14.1. Global Variables with dynamic names

Usually an assignment is enough to access and set a value. a global variable. However, we often need the option metaprogramming when we want to work with global re- variable whose name is contained in another variable or computed is done in the course of work. To get the value of such a variable, many some programmers try to use something like the following snippet of code:

```
value = loadstring (“return“ .. varname) ()
```

If varname is x , then as a result of concatenation we get "Return x" , which when executed will give us the desired result. Od-

but this code involves creating and compiling a new block code, which is costly. You can achieve the same go using the following code, which is more than an order of magnitude more more efficient than the previously discussed code:

```
value = _G [varname]
```

Since the environment is a regular table, you can simply access it by key (variable name). Similarly you can also assign a value to a variable whose name computes- dynamically, using the code _G [varname] = value . However, be careful: some programmers are so happy about this opportunity ness that end up writing code like _G [“a”] = _ G [“var1”] , which is just a tricky option for a = var1 .

The generalization of the previous task is to use names fields in dynamic names such as “io.read” or “abcd” . However, if we write _G [“io.read”] , then we will definitely not get read field from table io . But we can write a getfield function , such that getfield (“io.read”) will return the expected value. This

a function is a loop that starts with _G and then
she sequentially iterates over the fields:

```
function getfield (f)
local v = _G -– start with global variable table
for w in string.gmatch (f, “[% w _] +”) do
v = v [w]
end
return v
end
```

We use the gmatch function from the string library to
to bypass all words in f (a word is a sequence of letters,
numbers and underscore).
The corresponding function for setting field values is
is more complicated. An assignment like abcd = v is equivalent to
to the following code:

```
local temp = abc
temp.d = v
```

That is, we have to extract the name without the last component and then process the last component separately. Function setfield You are a completes this and also creates helper tables in the path if they don't exist:

```
function setfield (f, v)
local t = _G -– we start with the table of global variables
for w, d in string.gmatch (f, “([% w _] +) (%.?)”) do
if d == "." then - not the last name?
t [w] = t [w] or {} -– creates a table if it does not exist
t = t [w]
- we get the table
else
–- last name
t [w] = v
- perform the assignment
end
end
end
```

The variable w stores the name of the field, and possibly the following after it the point is stored in the variable d 1 . If the name is not followed - there is a point, then this is the last name.
Using the previously discussed functions, the following code creates global table t , table tx and then assigns 10 txy :

```
setfield (“txy”, 10)
print (txy) -> 10
print (getfield (“txy”)) -> 10
```

14.2. Descriptions of global variables

In Lua, global variables do not need declarations. Although it is convenient
but for small programs, in large programs there is only one opera-
Chat can lead to hard-to-find bugs. but
we can change this behavior if desired. Since Lua
stores global variables in a regular table, then we can use
use metatables to change behavior when referring to
global variables.
The first approach simply tracks any calls to the missing
the following keys in the global table:

```
setmetatable (_G, {
__newindex = function (_, n)
error (“attempt to write to undeclared variable“ .. n, 2)
end,
__index = function (_, n)
error (“attempt to read undeclared variable“ .. n, 2)
end,
})
```

After executing this code, any attempt to address the wrong
existing global variable will cause
errors:

```
> print (a)
stdin: 1: attempt to read undeclared variable a
```

However, how are we going to declare global variables? One
an option is to use rawset which doesn't use
metamethods:

```
function declare (name, initval)
rawset (_G, name, initval or false)
end
```

(The ***or false*** construct is needed so that the global variable
got a value other than ***nil*** .)
A simpler option is to restrict the assignments to but-
global variables only inside functions, allowing
piling on the outer level of the block.
To check that the assignment occurs in the main block
ke, we need to use a debug library. Call debug.
getinfo (2, "S") returns a table whose what field says
whether the function that called the metamethod is the main block,

normal function or C-function. (We will discuss debug.getinfo in more detail in Chapter 24.) Using this function, we can rewrite the __newindex metamethod as follows:

```
__newindex = function (t, n, v)
local w = debug.getinfo (2, “S”). what
if w ~ = “main” and w ~ = “C” then
error (“attempt to write to undeclared variable“ .. n, 2)
end
rawset (t, n, v)
end
```

This new version also allows assignments in C code, since usually in this code the authors know what they are doing. To check that such a variable exists, we cannot just compare it to ***nil*** , because if it is ***nil*** , then the children to error. Instead, we use the rawget function , which doesn't use metamethod:

```
if rawget (_G, var) == nil then
- 'var' is undeclared
...
end
```

Now our approach does not allow global variables with a value Niemi ***nil*** , because they will automatically be considered neobyavlennym. But this is easy to fix. All we need is an additional naya table containing the names of the described variables. When calling In the metamethod, it checks against this table whether this variable is described. Similar code is shown in Listing 14.1. Now even assignments x = nil is enough to declare a global variable.

The cost of both solutions is extremely low. At the first decision, in normal operation, the metamethod is not called at all. At the second In the solution, metamethods can be called when the program is grows to a variable whose value is ***nil*** .

The standard Lua package contains the strict.lua module , which implements checking of calls to global variables, similar similar to the code we have considered. A good habit is use it when writing Lua code.

Listing 14.1. Checking global variable declarations

```
local declaredNames = {}
setmetatable (_G, {
__newindex = function (t, n, v)
if not declaredNames [n] then
local w = debug.getinfo (2, “S”). what
if w ~ = “main” and w ~ = “C” then
error (“attempt to write to undeclared variable“ ..n, 2)
```

```
end
declaredNames [n] = true
end
rawset (t, n, v) - do the actual set
end,
__index = function (_, n)
if not declaredNames [n] then
error (“attempt to read undeclared variable“ ..n, 2)
else
return nil
end
end,
})
```

14.3. Non-global environments

One of the problems with the environment is that it is global.
Any change to it affects all parts of your program. On-
example when you set metatable to control
global access, your entire program must follow the appropriate
appropriate policy. If you want to use a library,
which uses global variables without declaring them, then
you were unlucky.
In Lua, global variables don't have to be truly glob
ballroom. We can even say that Lua has no global re-
men. This may sound strange, since from the very beginning of the book
we used global variables. Obviously, Lua is very
tries to create the illusion of having global variables. Come on-
Let's see how Lua creates this illusion [2] .
Let's start with the concept of free names. *A free name* is not a name
tied to an explicit description, that is, it is not found inside the area
These are the actions of a local variable (or a ***for*** loop variable , or
parameter) with this name. For example, both names var1 and var2 are
free names in the next block:

```
var1 = var2 + 3
```

Unlike what was said earlier, the free name does not relate
refers to a global variable (at least not directly
but). Instead, Lua translates any free var name into _ENV.
var . Therefore, the previous block is equivalent to the following:

```
_ENV.var1 = _ENV.var2 + 3
```

But what is this new _ENV variable ? It cannot be glo-

point variable, otherwise we again return to the original sample-
leme. The compiler cheats again. I have already said that Lua will consider
Treats each block as an anonymous function. Actually Lua
compiles our source block into the following code:

```
local _ENV = <some value>
return function (...)
_ENV.var1 = _ENV.var2 + 3
end
```

That is, Lua compiles any block of code in the presence of
limited value named _ENV .
Usually, when we load a block of code, the load function is initialized
lyses this predefined value with a reference to the global oc-
rifle. Therefore, our original block becomes equivalent to
next block:

```
local _ENV = <the global environment>
return function (...)
_ENV.var1 = _ENV.var2 + 3
end
```

[2]
Note that this mechanism was one of those parts of Lua that included
Were from version 5.1 to version 5.2. The following discussion is specific to Lua
5.2 and very little applies to previous versions.
The result of all these assignments is that the var1 field from
the global environment gets var2 plus 3.
At first glance, this may seem a little confusing.
a way to work with the global environment. I will not argue
that this is the simplest way, but it achieves the flexibility that is difficult
but get a simpler implementation.
Before we continue, let's articulate how Lua 5.2 works.
works with global variables:

- Lua compiles any block using a value
_ENV .
- The compiler translates any free var name into _ENV.var .
- The load (or loadfile) function initializes the _ENV value
a reference to the global environment.

In the end, it’s not all that difficult.
Some are confused because they are trying to find some
the magic behind these rules. There is no magic here. In part
However, the first two rules are completely done by the compiler. Behind
except that _ENV is known to the compiler, it
is an ordinary variable. Except for compilation, _ENV

doesn't make any special sense in Lua [3] . Similarly translation from var to _ENV.var is just syntactic replacement without hidden meaning. In particular, after this translation, _ENV will refer to the _ENV variable , which is visible in this code snippet, is from the rules of visibility.

14.4. Using _ENV

In this section we will look at some of the ways to use the the flexibility that the _ENV variable brings . Please be aware that each of these examples should be run as a separate a powerful block of code. If you enter line by line in the interpreter, then each line becomes a separate block and gets its _ENV variable . To execute a piece of code as a separate block, you either need to run it as a file or in interactively put inside a ***do-end*** pair .

Since _ENV is an ordinary variable, we can assign read it and read it just like any other variable. Assigned vanie _ENV = nil prohibit any access to global variables on throughout the remainder of the block. This can be useful for control which variables your code uses:

```
local print, sin = print, math.sin
_ENV = nil
print (13)
-> 13
print (sin (13)) -> 0.42016703682664
print (math.cos (13)) - error!
```

Any assignment to a free name will result in a similar error.

We can explicitly call _ENV to bypass the locale. variable variables:

```
a = 13
- global
local a = 12
print (a) -> 12 (local)
print (_ENV.a) -> 13 (global)
```

Of course, the main use of _ENV is to change ok- tool used by the code snippet. Once you have changed your environment, all calls to global variables will be used use a new table:

- change the current environment to an empty table
_ENV = {}
a = 1 - create a field in _ENV
print (a)
-> stdin: 4: attempt to call global 'print' (a nil value)

If the new environment is empty, then you lose access to all glo-
ball variables including print . Therefore, you first need to
fill it with some useful values, like the old
environment:

a = 15
- create a global variable
_ENV = {g = _G} - change the current environment
a = 1
- create a field in _ENV
g.print (a)
-> 1
g.print (ga) -> 15

Now when you access the "global" g , you get sta
Swarm environment in which there is a print function .
We can rewrite the previous example using the name _G
instead of g :

a = 15
- create a global variable
_ENV = {_G = _G} - change the current environment
Using _ENV
a = 1
- create a field in _ENV
_G.print (a)
-> 1
_G.print (_G.a) -> 15

For Lua, the name _G is the same name as everyone else. His from-
the only characteristic is that when Lua creates a global
score table, then it assigns it to a variable named _G . For
Lua doesn't care about the current value of this variable. But it is usually accepted
use the same name when we refer to the global
variable, as we did in the rewritten example.
Another way to populate your new environment is by inheritance
nie:

a = 1
local newgt = {} - create a new environment
setmetatable (newgt, {__index = _G})
_ENV = newgt
- install it
print (a) -> 1

In this code, the new environment inherits print and a from the old environment. zheniya. However, any assignment goes to the new table. Thereby
there is no danger of mistakenly changing the global environment, although it can still be changed via _G :

```
- continue the previous code
a = 10
print (a) -> 10
print (_G.a) -> 1
_G.a = 20
print (_G.a) -> 20
```

Since _ENV is a regular variable, it obeys
the usual rules of visibility. In particular, the functions defined
inside a block, refer to _ENV just like any other external
her variable:

```
_ENV = {_G = _G}
local function foo ()
_G.print (a)
- compiles to '_ENV._G.print (_ENV.a)'
end
a = 10
- _ENV.a
foo ()
-> 10
_ENV = {_G = _G, a = 20}
foo ()
-> 20
```

If we define a new local variable named _ENV ,
then access to free names will go through it:

```
a = 2
do
local _ENV = {print = print, a = 14}
print (a) -> 14
end
print (a) -> 2 (back to original _ENV)
```

Therefore, it is not difficult to build a function with its own (private) environment:

```
function factory (_ENV)
return function ()
return a
- “global” a
end
end
f1 = factory {a = 6}
f2 = factory {a = 7}
```

```
print (f1 ())
-> 6
print (f2 ())
-> 7
```

The factory function creates simple closures that return-
the value of local variables a. When the closure is created, then
the visible variable _ENV is the parameter _ENV from the containing function
tion factory ; so the closure uses this variable to add
a stupa for free names.
Using normal visibility rules, we can work with
environments in various ways. For example, we may have
several functions with a common environment for them or a function that
Toraya changes the environment in common with other functions.

14.5. _ENV and load

As I mentioned, load usually initializes the _ENV value
the loaded block is a pointer to the global environment. However,
load has an optional fourth parameter that specifies the value
reading for _ENV . (The loadfile function also has a similar para-
meter.)
As an example, let's say we have a typical configuration
a file that defines the various constants and functions used
washed by the program; it could be something like:

```
- file 'config.lua'
width = 200
height = 300
...
_ENV and load
```

We can load it with the following code:

```
env = {}
f = loadfile (“config.lua”, “t”, env)
f ()
```

All code from the config file will be executed with empty
env environment . More importantly, all of its definitions will be named
but into this environment. The config file cannot affect
anything else, even by mistake. Even malicious code cannot cause
thread a lot of harm. It can perform a DoS attack by wasting CPU time

and memory, but nothing else.
Sometimes you may need to execute a block multiple times,
each time with a different environment table. In this case, additional
the load argument doesn't help us. Instead, we have two
option.
The first option is to use the debug.setupvalue function
from the debug library. As the name implies, setupvalue poz-
tells us to change any incoming value (upvalue) of the given
functions. The following code illustrates its use:

```
f = loadfile (filename)
...
env = {}
debug.setupvalue (f, 1, env)
```

The first argument when calling setupvalue is a function, the second is
this is the index of the value, and the third is the new value. For our use
use of the second argument is always one: when the function
is the result of load or loadfile , Lua guarantees that there will be
only one value and that value is _ENV .
A small disadvantage of this solution is the dependence
from the debug library. This library breaks some
standard assumptions about programs. For example debug.
setupvalue violates Lua's visibility rules, which guarantee
that the variable cannot be seen outside of its scope
visibility.
Another way to run a block with different environments is
There is a slight change in the block when it is loaded. Imagine,
that we add the following line to the beginning of the loaded block:

```
_ENV = ...;
```

Recall from Section 8.1 that Lua treats any block as
function with variable number of arguments. Therefore, this line attaches
assigns the first block argument to _ENV , setting it
as an environment. After loading the block, we call the resulting
function, passing the desired environment as the first argument. Next
The following code snippet illustrates this idea using the function
loadwithprefix from Exercise 8.1:

```
f = loadwithprefix ("local _ENV = ...;", io.lines (filename, "* L"))
...
env = {}
f (env)
```

Exercises

Exercise 14.1. The getfield function that we have defined at the beginning of this chapter provides too little role, since it allows fields such as math? sin or string !!! gsub .Rewrite it to handle only one dot as a separator. (For this exercise (you may need the information in Chapter 21.)

Exercise 14.2. Explain in detail what happens next program and what its output will be.

```
local foo
do
local _ENV = _ENV
function foo () print (X) end
end
X = 13
_ENV = nil
foo ()
X = 0
```

Exercise 14.3. Explain in detail what happens next program and what its output will be.

```
local print = print
function foo (_ENV, a)
print (a + b)
end
foo ({b = 14}, 12)
foo ({b = 10}, 1)
```

Exercises

Chapter 15
Modules and packages

Lua does not usually establish any conventions. Instead of this Lua provides mechanisms that are powerful enough for groups developers to implement the conventions that suit them.

However, this approach does not work well for modules. One of the main the purpose of the module system is to allow different people to share locally use the code. The lack of a common policy prevents this sharing.

Since version 5.1, Lua has defined a set of conventions for mod-
leys and packages (a package is a collection of modules). These agreements are not required
any additional features from the language; programmer
you can implement them using what we have already seen in the language:
tables, functions, metatables and environments. Programmers can
use other agreements. However, other agreements may
lead to the fact that it will not be possible to use other people's modules and your modules cannot be used in other people's programs.
From the user's point of view, a *module* is some code (in Lua
or in C), which can be loaded with require and co-
which creates and returns a table. Anything the module exports
whether it be functions or tables, it defines inside this table,
which acts as a namespace.
For example, all standard libraries are modules. You can
use math library like this:

```
local m = require “math”
print (m.sin (3.14))
```

However, a separate interpreter (available as a command
strings) preloads all standard libraries with
code equivalent to the following:

```
math = require “math”
string = require “string”
...
```

This download allows us to use the normal math.sin notation .
The obvious advantage of using tables to implement
dul is that we can work with modules in the same way as
with tables, and use the power of Lua to do so. In most
In two languages, modules are not first class values (i.e.
they cannot be stored in variables, passed as arguments
functions, etc.), so these languages need special mechanisms
terms for every opportunity they want to offer for
modules. In Lua, you get these features for free.
For example, there are several ways to call a function from
module. The usual way is as follows:

```
local mod = require “mod”
mod.foo ()
```

User can set any local name for the module:

```
local m = require “mod”
m.foo ()
```

Alternative names can also be provided for individual

functions:

```
local m = require “mod”
local f = mod.foo
f ()
```

The nice thing about these features is that they don't require special support from the language. They only use what the language already provides.

A common complaint about require is that this function This does not allow passing an argument to the loaded module. For example measures, the mathematical module could receive an argument that allows to choose between using degrees or radians:

```
- bad code
local math = require (“math”, “degree”)
```

The problem is that one of the main objectives require yav- Avoid loading an already loaded module. As soon as possible the muzzle is loaded, it will be reused by any part of the program, who needs it. Therefore, when using the parameters there would be a problem if the same module was needed, but with other parameters:

```
- bad code
local math = require (“math”, “degree”)
- somewhere else in the same program
local math = require (“math”, “radians”)
```

In case you really want your module to be held parameters, it is better to create an explicit function to set them niya:

```
local mod = require ”mod”
mod.init (0, 0)
```

If the initializing function returns the module itself, then we we can write code like the following:

```
local mod = require ”mod” .init (0, 0)
```

Another option is to make the module return a function for initialization and already this function would return a table muzzle:

```
local mod = require ”mod” (0, 0)
```

In any case, remember that the module is loaded only once; the module itself must resolve initializations with conflicts.

15.1. Require function

The require function tries to minimize the assumptions about

what is a module. For require, a module is just some-
then the code that defines some values (such as functions
or tables containing functions). Usually this code returns
a table containing the functions of this module. However, since
this is done by the code of the module itself, not require , some modules
may choose to return different values or even have sideways
effects.
To load a module, we simply call require " *modname* " .
The first step of require is to check against the package.
loaded to see if this module is already loaded. If so, then require
returns the corresponding value. Therefore, as soon as the module
loaded, other calls requiring this module to be loaded are simply
will return the same value without executing any code.
If the module has not yet been loaded, then require looks for a Lua file with the name
it module. If it finds such a Lua file, then it loads it when
help loadfile . The result of this is the function that we
called the *bootloader* . (The bootloader is a function that, when called
ve returns a module.)

If require cannot find a Lua file with a module name, then it
looks for a C library with a module name. If she finds the right
the current C library, it loads it using package.
loadlib (which we discussed in section 8.3) and looks for a function named
It luaopen_ *modname* 1 . In this case, the bootloader is the result
volume loadlib , that is, a luaopen_ *modname* function that looks like
function in Lua.
Whether the module is a Lua file or a lib
library in C, require now has a loader for it. For windows-
When the module is loaded, require calls the loader with two
arguments: the name of the module and the name of the file with the loader.
(Pain-
Most modules simply ignore these arguments.)
chick returns some value, then require returns this value
read and store it in the package.loaded table in order to
always return exactly this value for this module. If for-
the loader does not return anything, then require behaves the same as if
the module would return ***true*** . Without this clarification, subsequent calls
require would execute this module again.

To force require to load the specified module,
times, we simply erase the entry for this module from the package.
loaded :
package.loaded. < *modname* > = nil
The next time you need this module, the require prode-
barks all the necessary work again.

Renaming a module

We usually use their original name as the module name.
names, but sometimes we have to rename a module to avoid
name conflict. A typical situation is loading different
versions of the same module, for example for testing. Modu-
do not have hardcoded names inside them, so usually
just rename the corresponding .lua file. but
we cannot edit the binary library to change
the name of its function luaopen_ * . In order to maintain similar
new renames, there is a little trick inside require : if
the module name contains a minus, then require strips off the part of the name up to
before the minus sign when creating the function name luaopen_ * . For example
measure, if the module name is ab , then require expects the corresponding
the next function will be named luaopen_b , not luaopen_a-b
(which wouldn't be a valid name in C anyway). therefore
if we need to use two modules named mod , then we can
Let's rename one of them to v1-mod , for example. When we call
wem m1 = require ”v1-mod” , require will find the renamed file
v1-mod and inside this file will find a function named luaopen_mod .

Search along the path

When searching for a file in Lua, require uses a search path that
ry slightly differs from the usual search paths. Typical
path is a list of directories where to search for the given file. One-
co in ANSI C (the abstract platform that Lua runs on)
there is no concept of a directory. Therefore the path used by require is
a list of *templates* , each of which specifies its own way of converting
setting the module name (the require argument) to the file name. More accurately,
each pattern in the path is a filename containing optional

question marks. For each pattern, require replaces every '?' on
module name and checks if there is a file with the corresponding name;
if not, it moves to the next template. Patterns out of the way
separated by semicolons (a character rarely used in
filenames on modern operating systems). For instance,
if the path is

?;?. lua; c: \ windows \?; / usr / local / lua /? / ?. lua ,

then the call to require ("sql") will try to open the following files:

sql
sql.lua
c: \ windows \ sql
/usr/local/lua/sql/sql.lua

The require function uses as special characters
only semicolon (as component separator) and question mark-
ny sign; everything else, including path delimiters and extensions
files is defined by the path itself.
The path that require uses to find files in Lua is
always the current value of the package.path variable . When starting Lua
it initializes this variable with the value of the next variable-
environment LUA_PATH_5_2 . If this environment variable is not
is installed, then Lua tries to use the environment variable
named LUA_PATH . If both are undefined, then Lua uses
the default path, set at compile time [2] . When using
using environment variables, Lua substitutes the default path
instead of any substring ";;" ... For example, if LUA_PATH_5_2
is equal to "mydir / ?. lua ;;" then the final path will be pattern
"Mydir / ?. lua" followed by the default path.
The C library search path works the same way, but
cpath is taken from the variable package.cpath (instead of package.
path). Similarly, this variable gets its initial value
from the environment variable LUA_CPATH_5_2 or LUA_CPATH . Typical
the value for UNIX systems is

./?.so;/usr/local/lib/lua/5.2/?.so

Note that the path defines the file extension. Pre-
The previous example uses .so for all templates; in Windows t-
The pictorial template will look like the following:

. \ ?. dll; C: \ Program Files \ Lua502 \ dll \ ?. dll

The package.searchpath function implements all of these conventions for
search for libraries. It gets the module name and path and looks for the file,
following the rules described above. It returns either the name of the first

found file, or ***nil*** and an error message describing all
the files she tried to open, as in the following example:

```
> path = ". \\ ?. dll; C: \\ Program Files \\ Lua502 \\ dll \\ ?. dll"
> print (package.searchpath ("X", path))
nil
no file '. \ X.dll'
no file 'C: \ Program Files \ Lua502 \ dll \ X.dll'
```

File crawlers

In reality, require is somewhat more complex than we have described.
Finding a file in Lua and finding a C library are just two private
the more general concept *of a file* searcher. Seeker
file is just a function that takes the name of the module and returns
sets a bootloader for this block, or ***nil*** if it cannot find any
one.
The package.searchers array contains a list of file searchers,
which require . When looking for a module, require calls
each crawler in turn, passing it the module name, until

[2] In Lua 5.2, the -E command line parameter prevents the use of variables environment and causes the compile-time path to be used.

until it finds a loader for the module. If the search ends in
that is, require raises an error.
Using a list to control the search for a module gives
more flexibility of the require function . For example, if you want to store
thread the modules compressed into zip files, then all you need to do this is -
this is to provide the appropriate finder function and add it
to the list. However, more often than not, programs do not need to change
package.searchers value . The default configuration is
the Lua function finder and the C library crawler we described
above, occupy the second and third positions. Before them there is a claim
preload searcher.
This finder allows you to enter an arbitrary function for
module load. It uses the package.preload table to match
setting the names of the modules of the boot functions. When searching for this
the crawler simply looks for the given name in the table. If he finds a function
tion, then it returns it as a module loader. Otherwise it returns
nil . This finder provides a way to handle some non-

typical cases. For example, a C library statically prilinked
bathroom to Lua, can register its luaopen_ function like this,
that it will be called only when (and if) the user needs
this module is beating. Thus, the program does not waste time on
opening a module if not in use.
By default package.searchers includes the fourth
function that is needed for submodules. We will consider them at a time
case 15.4.

15.2. Standard Approach for writing modules in Lua

The simplest way to create a module in Lua is really simple: we
create a table, put all the functions we want to export
tite, inside it and return this table. Listing 15.1 demonstration
redefines this approach. Note how we define the function
tion inv as a closed, simply declaring it inside the unit.
Some people don't like the terminating return statement . One of
ways to eliminate it is to write the module table directly
in package.loaded :

```
local M = {}
package.loaded [...] = M
<as before>
```

Be aware that require calls the loader passing the name of the mo-
blowing as the first argument. Therefore, the variable number expression
arguments ... results in exactly that name. After that, when
piles, we no longer need to return M at the end of the module: if
the module does not return a value, then require will return the current value
package.loaded [modname] (if not ***nil***). However, I prefer
return a table as it looks neater.
Another way to write a module is to define all the functions
localization and building the table at the end, as in Listing 15.2.
What are the advantages of this approach? You don't have to start every
name with M. or something similar; there is an explicit export list
functions; you define and use exported and
internal functions are exactly the same inside a module. In what
are the disadvantages of this approach? Exported List
functions is at the end of the module, not at the beginning where it would be

more convenient as a quick reference; and a list to export from-
is accurate, since you need to write down each name twice. (This last
this disadvantage can be an advantage, since it allows
functions have different names outside the module and inside it, but I
I think that programmers rarely use this.) I personally like-
This style is being developed.
However, remember that no matter how you define the
dul, users should be able to use it
in a standard way:

```
local cpx = require “complex”
print (cpx.tostring (cpx.add (cpx.new (3,4), cpx.i)))
-> (3.5)
```

Listing 15.1. Simple module for complex numbers

```
local M = {}
function M.new (r, i) return {r = r, i = i} end
- define the constant 'i'
Mi = M.new (0, 1)
function M.add (c1, c2)
return M.new (c1.r + c2.r, c1.i + c2.i)
end
function M.sub (c1, c2)
return M.new (c1.r - c2.r, c1.i - c2.i)
end
function M.mul (c1, c2)
return M.new (c1.r * c2.r - c1.i * c2.i, c1.r * c2.i + c1.i * c2.r)
end
local function inv (c)
local n = cr ^ 2 + ci ^ 2
return M.new (cr / n, -ci / n)
end
function M.div (c1, c2)
return M.mul (c1, inv (c2))
end
function M.tostring (c)
return “(“ .. cr .. “,” .. ci .. “)”
end
return M
```

Listing 15.2. Module with an explicit list of exported functions

```
local function new (r, i) return {r = r, i = i} end
- define the constant 'i'
local i = complex.new (0, 1)
<other functions follow the same pattern>
return {
new = new,
i = i,
add = add,
sub = sub,
```

```
mul = mul,
div = div,
tostring = tostring,
}
```

15.3. Using environments

One of the disadvantages of the considered methods for creating modules is that it is very easy to clog up the global namespace, for example just forgetting `local` in the local resource description. Environments provide an interesting approach to creating muzzle that solves this problem. If the module has its own environment living, then not only all functions will fall into this table, but also all global variables. Therefore, we can determine everything from- covered functions as global, and they will automatically fall into co- the corresponding table. All a module needs to do is assign this table to `_ENV` . After that, when we determine we divide the `add` function , it automatically becomes `M.add` :

```
local M = {}
_ENV = M
function add (c1, c2)
return new (c1.r + c2.r, c1.i + c2.i)
end
```

Moreover, we can call other functions from this module. without any prefix. In the previous code, `add` refers to `new` from his environment, that is, he actually refers to `M.new` . This method is a good way to create modules that require buoy of very little work from a programmer. Prefixes in general Not needed. There is no difference between calling the exported and closed functions. If the programmer forgets to insert `local` , then it doesn't litter the global namespace; instead closed the function just becomes exportable.

However, I usually prefer one of the two previously considered new methods. Although they may require a little more work, the at least the code is clearer. In order not to create a global value by mistake, I just assign `_ENV a` value ***nil*** . After that, any attempt to create a global value simply raises an error.

What is missing is access to other modules. After

after we changed the value of _ENV , we lost access to all
the previous global variables. There are several ways to
there is this access, each with its own pros and cons.
One option is to use inheritance:

```
local M = {}
setmetatable (M, {__index = _G})
_ENV = M
```

(You need to call setmetatable before assigning _ENV , so
is it clear why?) When using this approach, the module receives
direct access to any global variable, with a very small
the price of such access. An interesting consequence of this decision is
It means that your module now contains all the global variables.
For example, someone using your module can now call
a standard function for computing sine with complex.
math.sin (x) . (A similar feature is also present in the Perl language.)
Another quick way to access other modules is to enter
dividing a local variable containing the global environment:

```
local M = {}
local _G = _G
_ENV = M - or _ENV = nil
```

You should now start each global name with _G. but before-
stupas happen a little faster as there is no use of me-
there methods.

A more rigorous approach is to define as local
variables of only those functions or modules that you need:

```
- module setting
local M = {}
- import section:
```

```
- take outside everything this module needs
local sqrt = math.sqrt
local io = io
- from this place access to the outside is impossible
_ENV = nil - or _ENV = M
```

This approach requires more work, but it clearly documents
the dependencies of your module. It also leads to code that you-
is filled a little faster than in the previously considered cases, due to
for using local variables.

15.4. Submodules and packages

Lua allows for hierarchical module names using
touch the point to separate the levels. For example, a module named mod.
sub is a *submodule* of mod . *A package* is a complete tree of mo-
muzzle; it is the Lua distribution unit.

When you need a module named mod.sub , require first
looks in the package.loaded table and then in the package.preload table ,
using the fully qualified name "mod.sub" as the key; in this case, exactly
ka is the same symbol as any other.

However, when looking for a file defining this submodule, require re-
converts point to another character, usually the system separator in the path
(i.e. '/' for UNIX and '\' for Windows). After this transformation
require looks for the resulting name just like any other name. For instance,
path '/' is a path separator and we have the following path:

```
./?.lua;/usr/local/lua/?.lua;/usr/local/lua/?/init.lua
```

The require ("ab") call will try to open the following files:

```
./a/b.lua
/usr/local/lua/a/b.lua
/usr/local/lua/a/b/init.lua
```

This behavior allows all modules in the package to be in
useful catalog. For example, if a package contains p modules ,
pa and pb , the corresponding files can be p / init.lua ,
p / a.lua and p / b.lua , where the p directory is contained in the corresponding
location
those.

The path separator used by Lua is set at compile time.
tion and can be any string (remember that Lua knows nothing
about directories). For example, systems without hierarchical catalogs can

gotta use '_' as such a delimiter, so require
("Ab") will look for a_b.lua file .
Names in C cannot contain periods, so the C library
for a submodule ab cannot export the luaopen_a.b function .
In this case, require translates point into another character - underscore-
vanie. Thus, a C library named ab should call
its initializing function luaopen_a_b . We can also
use here a technique with a minus, but with a more complex result-
Tom. For example, suppose we have a C library named a and we
want to make it a submodule mod , then we can rename
the corresponding file in mod / va . When calling require "mod.va"
the require call will correctly find the new mod / va file , just like
function luaopen_a inside it.
Also require has one extra crawler to load
submodules in C. When it cannot find either the Lua file or the C file for
submodule, this seeker again searches along the path for C, but this time seeks
package name. For example, if the program wants to load a submodule
abc , then this searcher will simply search for a . If he finds a library
I'm fluent in C for this name, then require will look in that library
the corresponding function, in our case luaopen_a_b_c . This charge
the ability to place several submodules together in one
a C library, each with its own initializing function.
From a Lua perspective, submodules in the same package do not have an explicit
communication. Loading module a does not load any of its sub-
modules; also loading ab doesn't load automatically a . End-
but, when implementing the package, the developer has the right to set these
links when
desire. For example, module a may explicitly require loading as
someone specific (or all) of their submodule.

Exercises

Exercise 15.1. Rewrite the code in Listing 13.1 as separate
module.

Exercise 15.2. What happens when looking for a library if the path
contains a fixed component (that is, a component not
containing a question mark)? Could this behavior be
useful?

Exercise 15.3. Write a finder that simultaneously looks for files in Lua and libraries in C. For example, the path for this searcher could be something like:

```
./?.lua;./?.so;/usr/lib/lua5.2/?.so;/usr/share/lua5.2/?.lua
```

(*Hint* : use package.searchpath to search the corresponding file, then try to download it, first with loadfile , then with package.loadlib .)

Exercise 15.4. What happens if you set a metatable for package.preload using the __index metamethod ? Mocan this be useful?

Chapter 16
Object oriented programming

A table in Lua is an object in more than one sense. Like objects, the table has a state. Like objects, a table has there is an identity (self) that does not depend on its values; in partness, two tables with the same values are different objects, an object can have different meanings at different times time. Like objects, tables have a life cycle that does not depend on who created them or where they were created. Objects have their own methods. Tables can also have their own todes as below:

```
Account = {balance = 0}
function Account.withdraw (v)
Account.balance = Account.balance - v
end
```

This definition creates a new function and stores it in the field withdraw of the Account object . Then we can call her as shown below:

```
Account.withdraw (100.00)
```

A function of this type is almost what we call a *method* . However, using the global Account name inside a function is bad practice. First, this feature will work only for this particular object. Secondly, even for this

object exactly as long as this object is recorded in this particular
a global variable. If we change the name of the object, then withdraw
will no longer work:

```
a, Account = Account, nil
a.withdraw (100.00) - ERROR!
```

This behavior violates the principle that every object
must have its own, independent cycle of life.
A more flexible option is to use the *recipient of the* opera-
walkie-talkies. To do this, our method will need an additional argument
cop with the value of the recipient. This parameter is usually named *self*
or *this* :

```
function Account.withdraw (self, v)
self.balance = self.balance - v
end
```

Now, when we call the method, we must indicate with which
object it should work:

```
a1 = Account; Account = nil
...
a1.withdraw (a1, 100.00) - OK
```

When using the *self* parameter, we can use one and
the same method for many objects:

```
a2 = {balance = 0, withdraw = Account.withdraw}
...
a2.withdraw (a2, 260.00)
```

This use of the *self* parameter is key in any design.
an object-oriented language. Most object-oriented
in languages, this mechanism is partially hidden from the programmer,
therefore, this parameter does not need to be explicitly declared (although inside the method
you can still use - *self* or *this*). Lua can also
hide this parameter with the *colon* e *operator* . We can
rewrite the previous method definition as follows:

```
function Account: withdraw (v)
self.balance = self.balance - v
end
```

Then the method call will look like this:

```
a: withdraw (100.00)
```

The colon adds an extra hidden parameter in the definition
method and adds an extra argument to the method call.
The colon is just syntactic sugar, albeit pre-
quite comfortable; there is nothing fundamentally new here. We can

define a method when using dot syntax and call
it using colon syntax, and vice versa, as long as
we handle the extra parameter correctly:

```
Account = {balance = 0,
withdraw = function (self, v)
self.balance = self.balance - v
end
}
function Account: deposit (v)
self.balance = self.balance + v
end
Account.deposit (Account, 200.00)
Account: withdraw (100.00)
```

At this point, our objects have an identity, consisting
and operations on this state. They lack a class system
inheritance and the ability to hide your variables (state).
Let's deal with the first task first: how can we create
different objects with the same behavior? Like how we
can we create multiple accounts?

16.1. Classes

The class acts as a template for creating objects. Most
object oriented languages offer the concept of a class.
In such languages, each object is an instance of some
specific class. Lua has no concept of a class; each object is defined
shares his behavior and his data. However, it is not difficult at all -
emulate classes in Lua, following the path of prototype languages like
Self or NewtonScript. Objects have no classes in these languages. Inmes-
then each object can have a prototype that is
the object in which the first object is looking for operations that it is not
knows. To represent classes in such languages, we simply create
an object that will only be used as a prototype for
other objects (its instances). Both classes and prototypes will
fall as a place to accommodate behavior common to
various objects.
In Lua, we can implement prototypes using the idea of inheritance
from section 13.4. More precisely, if we have two objects `a` and `b` , then

everything
what we need to do to make b act as a prototype for a is
following:
```
setmetatable (a, {__index = b})
```
After that, a will search in b for all operations that it does not know.
Setting b as a class for a is actually practically
same.
Let's go back to our bank account example. For co-
building other accounts with behavior similar to Account , we will make
so that these new objects will inherit their operations from Account when
using the __index metamethod . As a small optimization, we
we can not create separate metatables for each of the objects;
instead, we'll use the Account table itself :
```
function Account: new (o)
o = o or {} — create a table if the user has not submitted it
setmetatable (o, self)
self .__ index = self
return o
end
```
(When we call Account: new , self is equal to Account ;
so we could explicitly use Account instead of self . Od-
But using self is very useful for us in the following
section when we introduce inheritance.) What happens when we
create a new account and call its method as shown below?
```
a = Account: new {balance = 0}
a: deposit (100.00)
```
When we create a new account, a will have an Account (pa-
the *self* parameter when invoking Account: new) as a metatable. Then,
when we call a: deposit (100.00) we are actually calling
a.deposit (a, 100.00) ; the colon is just syntactic sugar.
However, Lua cannot find the deposit record in table a ; so Lua
looks for an __index entry in the metatable. The situation looks like
in the following way:
```
getmetatable (a) .__ index.deposit (a, 100.00)
```
Metatable a is Account and Account .__ index is so-
the same Account (since the new method did self .__ index = self).
Therefore, the previous expression is reduced to
```
Account.deposit (a, 100.00)
```
That is, Lua calls the original deposit function , but passing
a as a parameter to self . Thus, the new account a inherited
shaft deposit function from Account . In the same way, he inherits everything

fields from Account .
Inheritance works not only for methods, but also for other
some fields that are not in the new account. Therefore, the class can
set not only methods, but also default values for fields
instance. Recall that in our first definition of Account
we provided the balance field with a value of 0. So if we match
create an account without the initial balance value, then it will inherit this
default value:

```
b = Account: new ()
print (b.balance) -> 0
```

When we call b's deposit method , this call will be equivalent to
taped to the following code (since *self* is b):

```
b.balance = b.balance + v
```

Expression b.balance gives 0, and the method assigns an initial
b.balance contribution . Subsequent calls to b.balance no longer apply.
lead to a call to the corresponding metamethod, since b now has
your balance field .

16.2. Inheritance

Since classes are objects, they can also receive
methods from other classes. This behavior makes it easy to implement
inherit (in the usual object-oriented sense
le).
Let's say we have a base class Account :

```
Account = {balance = 0}
function Account: new (o)
o = o or {}
setmetatable (o, self)
self .__ index = self
return o
end
function Account: deposit (v)
self.balance = self.balance + v
end
function Account: withdraw (v)
if v> self.balance then error ”insufficient funds” end
self.balance = self.balance - v
end
```

From this class we can inherit the SpecialAccount class ,
allowing the buyer to withdraw more than is on his balance sheet. we
starting with an empty class that inherits all operations from its

base class:
SpecialAccount = Account: new ()

Up to this point, SpecialAccount is just an instance
Account . However, interesting things happen next:
s = SpecialAccount: new {limit = 1000.00}
SpecialAccount inherits new from Account , like all other methods.
Toda. However, this time when new is executed, its self parameter is already
will refer to SpecialAccount . Therefore, the metatable s will be
SpecialAccount , whose __index is equal to SpecialAccount .
Therefore, s inherits from SpecialAccount , which in turn
inherits from Account . Now if we do
s: deposit (100.00) ,
then Lua cannot find the deposit field in s , so it will look for it
in SpecialAccount , there he will not find it either and will continue to search in
Account , where it will find the original implementation of this method.
What makes SpecialAccount special is that we can re-
define any method inherited from its parent class
sa. All we need is to simply write a new method:

```
function SpecialAccount: withdraw (v)
if v - self.balance> = self: getLimit () then
error ”insufficient funds”
end
self.balance = self.balance - v
end
function SpecialAccount: getLimit ()
return self.limit or 0
end
```

Now, when we call s: withdraw (200.00) , Lua will not return-
in Account , since it will find a new withdraw method before
in the SpecialAccount class . Since s.limit is 1000.00 (we set
this field when creating s), then the program will remove it, leaving
resulting in s with negative balance.
The interesting thing about objects in Lua is that you don't
you need to create a new class to define the new behavior. If from-
it is only necessary to change the behavior for one object, then we can re-
lize this change directly in this object. For instance,
if account s represents a special customer whose limit is always 10%
from the current balance, then we can change only one account:
function s: getLimit ()

```
return self.balance * 0.10
end
```

After that, calling s: withdraw (200.0) will execute the withdraw method from the SpecialAccount class , but when withdraw calls s: getLimit , then the previously entered definition of this function will be called.

16.3. Plural inheritance

Since objects are not basic primitives, Lua has several ways to use object-oriented software gramming. The approach we just saw uses- the __index metamethod is probably the best combination simplicity, speed and flexibility. However, there are other implementations too, which may be more suitable for some specific lent cases. We will now see an alternative implementation, which Toraya allows multiple inheritance in Lua.

The key in this implementation is to use the function in as the __index metafield . Recall that when the metatable has data table has an __index field , then Lua will call this function any time when it cannot find the key in the source table. In this case __index can search for a missing key in any number of po- parents.

Multiple inheritance means that a class can have more than one superclass (parent class). Therefore, we already cannot use a function like before to create a sub- black classes. Instead, we will define a createClass function , which takes parent classes as arguments (see Listing 16.1). This function creates a table for the presentation new class and sets its metatable with metamethod __index , which implements multiple inheritance. Not- looking at multiple inheritance, each created object belongs to one class, which is used to find the place todes. Therefore, the relationship between class and superclasses is differs from the relationship between classes and its instance- mi (created objects). In particular, a class cannot simultaneously it should be the metatable for its instances and child classes.

In Listing 6.1, we use the class as the metatable for the generated

instances and create a separate table as a metatable
class.

Listing 16.1. Implementing multiple inheritance

```
- look for 'k' in the list of tables 'plist'
local function search (k, plist)
for i = 1, #plist do
local v = plist [i] [k] - try the i-th superclass
if v then return v end
end
end
function createClass (...)
local c = {} -– new class
local parents = {...}
- the class will search for each method in the list of its parents
setmetatable (c, {__index = function (t, k)
return search (k, parents)
end})
- prepare 'c' as a metatable of its instances
c .__ index = c
- define a new constructor for this new class
function c: new (o)
o = o or {}
setmetatable (o, c)
return o
end
return c - return a new class
end
```

Let's illustrate the use of createClass by
cabbage soup small example. Let's say we have our old Account class
and Class Named to methods setname and getname .

```
Named = {}
function Named: getname ()
return self.name
end
function Named: setname (n)
self.name = n
end
```

To create a new class NamedAccount , which is a child
with both Account and Named classes , we just call createClass :

```
NamedAccount = createClass (Account, Named)
```

We create and use instances of this class as before:

```
account = NamedAccount: new {name = “Paul”}
print (account: getname ()) -> Paul
```

Now let's see how the last statement works. Lua
cannot find getname method in account ; so he's looking for a field

__index in the account metatable , that is, in the NamedAccount . But in NamedAccount also doesn't have a " getname " field , so Lua looks for a field __index in the NamedAccount metatable. Since this field contains function, Lua calls it. This function first looks for "getname" in Account and, not finding it there, looks for Named , where she finds an excellent from ***nil the*** value that becomes the final result.
Of course, due to the complexity of such a search, the performance for many physical inheritance differs from performance for simple that inheritance. An easy way to improve this performance is to copy inherited methods into child classes.
Using this approach, the __index metamethod would look like in the following way:

```
setmetatable (c, {__index = function (t, k)
local v = search (k, parents)
t [k] = v - save for next call
return v
end})
```

Using this technique, access to inherited methods becomes as fast as accessing local methods (except for the first call). The disadvantage is that it is difficult to change method definitions when the system is working no, since these changes do not carry over along the chain of inheritance giving.

16.4. Hiding

Many consider the possibility of hiding to be an integral part of the object-but-oriented language; the state of each object is his personal matter. In some object oriented languages, such as C ++ and Java, you can control whether the field is an object one or his method is visible from the outside. In Smalltalk, all variables are hidden, and all methods are accessible from the outside. Simula, the first object-oriented language, does not provide such protection for lei and methods.
The Lua object design we looked at earlier did not pre-delivers hiding mechanisms. This is partly a consequence of

our use of tables to represent objects. Besides
Moreover, Lua avoids redundancy and artificial limitations. If a
you don't want to access the fields inside the object, just *don't*
this .

However, another goal of Lua is flexibility, it provides
provides meta-mechanisms that allow you to emulate many of the
nosti. Although basic object design for Lua does not provide
hiding mechanisms, we can implement objects in a different way
so that you gain access control. Although this possibility
programmers use infrequently, it will be useful to learn about
her, as it reveals some interesting aspects
Lua and can be a good solution for other tasks as well.
The main idea behind alternative design is to represent each
each object using two tables: one for its state and the other
gaya - for his operations (his interface). The object is being accessed
through the second table, that is, through the operations that form its in-
terface. In order to avoid unauthorized access,
the table providing its state is not stored in another field
goy table, it is only accessible through closures within methods.
For example, to represent a bank account using this
design, we will create new objects using the following
factory functions:

```
function newAccount (initialBalance)
local self = {balance = initialBalance}
local withdraw = function (v)
self.balance = self.balance - v
end
local deposit = function (v)
self.balance = self.balance + v
end
local getBalance = function () return self.balance end
return {
withdraw = withdraw,
deposit = deposit,
getBalance = getBalance
}
end
```

The function first creates a table to store the internal
state of the object and stores it in the local variable self .

The function then creates methods for the object. Finally, the function is
gives and returns an external object that matches the names of the
methods to their implementations. The key here is that these methods
don't get self as an optional parameter; instead they
access self . Since the additional
there is no argument, then we do not use the colon syntax for
work with the object. We call their methods just as usual.
functions:

```
acc1 = newAccount (100.00)
acc1.withdraw (40.00)
print (acc1.getBalance ()) -> 60
```

This design provides complete stealth for everything that is stored
is stored in the self . After returning from the newAccount function
there is no way to directly access this tab-
face. Although our example only stores one variable in a private
table, we can store all the private parts of the object in this table
face. We can also define private methods: they are like
public, but we don't put them in the interface. For example, our accounts
can provide an additional 10% loan with the balance
above a certain value, but we do not want the user to
whether they had access to the details of the calculations. We can implement this
functionality as follows:

```
function newAccount (initialBalance)
local self = {
balance = initialBalance,
LIM = 10000.00,
}
local extra = function ()
if self.balance> self.LIM then
return self.balance * 0.10
else
return 0
end
end
local getBalance = function ()
return self.balance + extra ()
end
<as before>
```

Again there is no way to call the function directly
extra .

16.5. Single approach method

A special case of the previous approach for object-oriented The case when the object has only one method. In this case, we do not need to create an interface tab- face; we can just return this method as a view object. If this looks a little strange, let's remember the time cases 7.1, where we created iterative functions that store their co- standing as closures. An iterator storing its state is nothing is no different from an object with a single function. Another interesting case of objects with a single method is the case when this method actually performs a different tasks depending on a specific argument. Possible the implementation of such an object is shown below:

```
function newObject (value)
return function (action, v)
if action == “get” then return value
elseif action == “set” then value = v
else error (“invalid action”)
end
end
end
```

Its usage is pretty simple:

```
d = newObject (0)
print (d (“get”)) -> 0
d (“set”, 10)
print (d (“get”)) -> 10
```

This object implementation is pretty efficient. Syntax
`d (“set”, 10),` although it looks strange, is only two characters long
it than the traditional `d: set (10)` . Each object uses one
closure, which is cheaper than one table. There is no inheritance here, but
but we have complete secrecy: the only way to contact
to the state of an object is to use its only
th method.

Tcl / Tk uses a similar approach for its widgets. Name kind-
a get in Tk denotes a function (*widget command*) that can
perform various types of operations on the widget.

Exercises

Exercise 16.1. Implement the Stack class with push , pop , top and isempty .
Exercise 16.2. Implement class StackQueue as a subclass Stack . In addition to the inherited methods, add to this class su method insertbottom , which inserts an element at the end stack. (This method allows you to use objects of the given class as a queue.)

Exercise 16.3. Another way to ensure that the public is closed projects is to implement them using a proxy (see section 13.4). Each object is represented by an empty table tsey (proxy). The internal table sets the corresponding vie between these empty tables and tables carrying the state of the object. This internal table is not available sleep guns, but the methods use it to translate their parameter self to the real table they are working with. Implement- the example with the Account class using this approach and see its pros and cons.
(There is one small problem with this approach. Try- you can find it yourself or refer to section 17.3 for suggestions her solution is proposed.)

Chapter 17
Weak tables and finalizers

Lua handles memory management. Programs create objects (tables, threads, etc.), but there is no function to destroy objects. Lua automatically destroys objects that become garbage rum, using *garbage collection* . This frees you from the main work with memory and, more importantly, frees most errors related to this activity, such as dangling links and memory leaks.
Using a garbage collector means Lua has no problem with cycles. You don't need any special action when using naming circular data structures; they automatically release

are given like any other data. However, sometimes even the clever
the garbage collector needs your help. No garbage collector
will allow you to forget about all the problems of resource management,
such as external resources.
Weak tables and finalizers are mechanisms that you
can be used in Lua to help the garbage collector.
Weak tables allow collection of Lua objects that are still pending
are accessible to the program, while finalizers allow assembly
external objects not under direct control
Lem the garbage collector. In this chapter, we will discuss both of these
mechanisms.

17.1. Weak tables

The garbage collector can only collect what is guaranteed to be
rubbish; he cannot guess for himself what is rubbish
your opinion. A typical example is a stack implemented
as an array, with a reference to the top of the stack. You know that the data is
reap only from the beginning of the array to this index (top of the stack),
but Lua doesn't know that. If you pop an item off the top of the stack,
simply decreasing the vertex index, then the object remaining in the array
is not garbage for Lua. Similarly, any object for which
the referenced global variable is also not garbage for
Lua even if you never use it. In both cases
you (more precisely, your program) should write **nil** in the appropriate
variables (or array elements) in order to avoid
the appearance of indestructible objects.
However, simply removing links is not always enough. In some
cases need additional interaction between your product
gram and garbage collector. A typical example is the set
all active objects of a certain type (for example, files) in your
our program. The task seems simple: all you need is
add every new object to this set. However, as soon as
the object becomes part of the collection, it will never be destroyed!
Even if no one refers to it, the set will still refer
on him. Lua cannot know that this link should not prevent
destroying that object, unless you tell Lua to do so.
Weak tables are the mechanism you use in

Lua to say that a link should not interfere
destruction of the object. *A weak reference* is such a reference to an object,
which is not counted by the garbage collector. If all links, indicate-
on an object are weak, then this object is free
and all these weak links are destroyed. Lua implements weak
links using weak tables: a *weak table* is such a
a person whose links are all weak. This means that if
the object is stored only inside a weak table, then the garbage collector
sooner or later will destroy this object.
Tables store keys and values, both of which can be object
tami of any type. Under normal conditions, the garbage collector is not
destroyed.
It also contains objects that are keys and references in an accessible
table. Both keys and values are *strong* references, i.e.
they prevent the destruction of those objects to which they point
call. In a weak table, both keys and values can be weak.
This means that there are three types of weak tables: tables with weak
strong keys, tables with weak values and completely weak
tables where both keys and values are weak. Regardless
type of table, when deleting a key or value, the entire record is deleted
is taken from the table.
The weakness of a table is specified by the __mode field of its metatable. Mean-
the nest of this field, when present, must be a string: if
this row is “k” , then the keys in this table are weak;
if this string is “v” , then the values in this table are weak
face; if this string is equal to "kv" , then both the keys and values in this table
face are weak. The next example, albeit artificial,
shows the behavior of weak tables:

```
a = {}
b = {__mode = “k”}
setmetatable (a, b) - now 'a' has weak keys
key = {}
- create the first key
a [key] = 1
key = {}
- create the second key
a [key] = 2
collectgarbage () - force the garbage collector to remove garbage
for k, v in pairs (a) do print (v) end
-> 2
```

In this example, the second assignment key = {} destroys the reference to
first key. Calling collectgarbage causes the garbage collector to delete
pouring all the trash. Since there are no more references to the first key, this
the key and the corresponding entry in the table are deleted. The second key is
is still stored in the variable key , so it is not deleted.
Please note that only objects can be removed from
weak table. Values such as numbers and booleans,
are not deleted. For example, if we insert a numeric key into table a
(from our previous example), then the garbage collector never has it
will delete. Of course, if the value corresponding to the numeric key is
is stored in a table with weak values, then all corresponding
the entire record is removed from the table.
There is a certain subtlety with lines: although lines are deleted-
by the garbage collector, in terms of implementation they differ from
other objects. Other objects such as tables and threads are co-
are given explicitly. For example, when Lua evaluates the expression {} , then it
creates a new table. However, does Lua create a new line when you
filling in "a" .. "b" ? What if the system already has the string “ab” ? Con-
will Lua give a newline? Can the compiler generate this line
before executing the program? It makes no difference: it
all implementation details. From a programmer's point of view, strings are
are values, not objects. Therefore, just like a number or a lo-
logical meaning, the row cannot be deleted from the weak table
(unless the associated value is removed).

17.2. Functions with caching

A common programming technique is to obtain
gain in time due to memory loss. You can speed up the function by caching its results so that when later you
call the same function with the same arguments, the function can
use the value stored in the cache. Imagine a server receiving requests as strings with
holding Lua code. Each time a request is received, the server executes load on the received line and then calls the received function. However, load is an expensive feature and some commands to the server can be repeated many times. Instead of constantly- call load every time the server receives a command like Closeconnection () , the server can remember the result of the load in auxiliary table.

The server checks before calling load , no whether there is already a value corresponding to the given string. If he can't
find the corresponding value, then (and only then) the server calls load and stores the result in this table. We can re- lize this behavior with the following function:

```
local results = {}
function mem_loadstring (s)
local res = results [s]
if res == nil then
- no result?
res = assert (load (s)) - calculate new result
results [s] = res
- save the result
end
return res
end
```

The gains from this scheme can be very significant. but
it can also cause large memory losses. Although some
mandas are repeated over and over, many other teams meet
just one time. Over time, the results table collects all the commands
which the server has ever received, and the corresponding code; since time
This can lead to memory exhaustion on the server. Weak tabs
faces provide a simple solution to this problem. If the table
results stores weak values, then each garbage collection cycle will remove
all currently unused values (virtually all):

```
local results = {}
setmetatable (results, {__mode = “v”}) - values will be weak
function mem_loadstring (s)
<as before>
```

In fact, since indices are always strings, we
we can make this table completely weak if we want to:

```
setmetatable (results, {__mode = “kv”})
```

The caching technique is also useful to ensure that
the caliber of objects of a certain type. For example, let us pre-
set colors as tables with red , green and blue fields . The simplest
the color factory will create a new table every time we
we turn to her:

```
function createRGB (r, g, b)
return {red = r, green = g, blue = b}
end
```

Using caching we can reuse tables

for the same colors. To create a unique key for
for each color, we simply connect the color components using
some separator:

```
local results = {}
setmetatable (results, {__mode = “v”}) -– values will be weak
function createRGB (r, g, b)
local key = r .. “-” .. g .. “-” .. b
local color = results [key]
if color == nil then
color = {red = r, green = g, blue = b}
results [key] = color
end
return color
end
```

An interesting consequence of this implementation is that the
the vendor can compare colors for equality using the standard
comparison operator, since two simultaneously existing
the same table will always correspond to the same colors.
Please note that the same color may be displayed differently.
tables at different points in time, since from time to time
The garbage collector will empty the results table . However, while
this color is used, it cannot be removed from results . therefore
if a color exists long enough to be compared with another
color, its representation will also exist for just as long.

17.3. Object attributes

Another interesting use of weak tables is linking
ding attributes with objects. There are an infinite number of situations
ation when we may need to bind some attribute
to object: names to functions, defaults to tables,
sizes to arrays, etc.

When the object is a table, then we can remember the attribute
in the table itself by choosing a suitable unique key. As we already
seen, a simple and reliable way to create a unique key is to create
give a new object (usually a table) and use it as
key. However, if the object is not a table, then this approach is already

not good. Even for tables, we may need to not store at-
ribut in the table itself. For example, we might want to do a similar
the attribute is private or we don't want to influence how the table
gets over. In all these cases, we need a different way of connecting
the use of attributes with objects.
Of course, a separate table provides an ideal way
binding attributes to objects (it is no coincidence that tables are different
yes they are called *associative arrays*). We can use
objects as keys, and their attributes as values. Such a table can
can store attributes of objects of any type, since Lua allows
use objects of any type as table keys. More
Moreover, attributes stored in a separate table do not affect other
objects and can be private, just like the table itself.
However, this solution has a huge disadvantage: as soon as
we used the object as a key in the table, it can no longer
be removed by the garbage collector. Lua cannot delete an object that
ry is used as a key. If we use the usual
table, in order to bind their names to functions, then none
of these functions will never be removed. How can you guess
live, we can avoid this deficiency with weak tab-
persons. However, this time we need weak keys. Use-
the creation of weak keys does not prevent the garbage collector from deleting these keys,
when there are no more links left. On the other hand, the table
there cannot be weak values; otherwise the attributes of existing objects
The objects could be deleted.

17.4. Again tables with default values

In section 13.4 we looked at how you can work with values by
default non- ***nil*** . We showed one approach and noticed
that the other two approaches require the use of weak tables, therefore
we have postponed the story about them for later. Now it's time to get back to
this topic. As you will see, these two approaches to implementing values by
defaults are in fact special cases already considered
well-known approaches, namely object attributes and caching.
In the first approach, we use weak tables to
bind the table to its default values:

```
local defaults = {}
setmetatable (defaults, {__mode = “k”})
local mt = {__index = function (t) return defaults [t] end}
function setDefault (t, d)
defaults [t] = d
setmetatable (t, mt)
end
```

If defaults did not use weak keys, then all tables with default values would always exist.

In the second solution, we use different metatables to define default values, but at the same time we reuse one and the same metatable when we use the same value again default. This is a typical case for caching:

```
local metas = {}
setmetatable (metas, {__mode = “v”})
function setDefault (t, d)
local mt = metas [d]
if mt == nil then
mt = {__index = function () return d end}
metas [d] = mt - remember
end
setmetatable (t, mt)
end
```

In this case, we use weak values in order to avoid the metatables used could be assembled by a mu-litter.

Which of the two is the best solution? As usual this depends on the use. Both solutions have approximately the same complexity and the same speed. First solution requires several words of memory for each table with a value by default (for writing to defaults). The second solution requires no how many tens of words of memory for each unique meaning default (new table, new closure plus metas entry).

Therefore, if your application has thousands of tables with only a few different default values, then the second solution is explicitly will be better. On the other hand, if several tables have a common with the default values, you'd better prefer the first implementation.

17.5. Ephemeral tables

An interesting case occurs when in a table with weak keys

the value refers to its own key.
This case is much more common than it might seem.
A typical example is a factory that returns functions.
A factory like this takes an object and returns a function that
will return this object when called:

```
function factory (o)
return function () return o end
end
```

This factory is a good candidate for caching, for
in order not to create new closures when there is already a suitable
the next already created closure:

```
do
local mem = {}
setmetatable (mem, {__mode = “k”})
function factory (o)
local res = mem [o]
if not res then
res = function () return o end
mem [o] = res
end
return res
end
end
```

However, there is one catch. Note that the value is
(corresponding function) associated with the object located in
mem , - refers to its own key (the object itself). Although the keys
are weak in this table, but values are not weak
Xia. With the standard interpretation of weak tables, nothing will
removed from the caching table. Since the values are not
weak, then there is always a strong reference to each function. Each
the function refers to its object, that is, there is always a strong reference
for each object. Therefore, these objects cannot be deleted, not
looking at the use of weak keys.
However, this interpretation is not always very helpful. Pain-
Most people expect the value in the table to be available only
through the appropriate key. Therefore, we can consider
a similar scenario as a loop case where the closure refers to
an object that (via the caching table) itself references this
closure.

Lua 5.2 solves this problem with ephemeral tables.
In Lua 5.2, a table with weak keys and strong values is

is *an* ephemeron table. In an ephemeral table
key availability controls the availability of the corresponding key
cheniya. Let's take a closer look at the entry (*k*, *v*) in an ephemeral table.
A reference to *v* is strong only if there is a strong reference to *k* .
Otherwise, the record is eventually deleted from the table, even
if *v* refers (directly or indirectly) to *k* .

17.6. Finalizers

Although the purpose of the garbage collector is to remove Lua objects, it
can also help the program free external resources. For
for these purposes, various programming languages offer mechanisms
finalizers. *The finalizer* is a function related to
ectom, which is called before the object is removed collection-
rubbish box.
Lua implements finalizers with the __gc metamethod . By-
look at the following example:

```
o = {x = “hi”}
setmetatable (o, {__gc = function (o) print (ox) end})
o = nil
collectgarbage () -> hi
```

In this example, we first create a table and set for
her a metatable that has a __gc metamethod . Then we destroy-
we get the only link to this table (global variable o)
and invoke garbage collection by calling collectgarbage . In
garbage collection time Lua detects that the given table is not
is available and calls its finalizer (__gc metamethod).
The subtle thing in Lua is marking an object for final
zation. We mark an object for finalization when we set it to
a metatable with a non-zero __gc field . If we do not mark the object, then
it will not be finalized. Most of the code we write is
will work, but sometimes weird cases like the following
blowing:

```
o = {x = “hi”}
mt = {}
setmetatable (o, mt)
mt .__ gc = function (o) print (ox) end
o = nil
collectgarbage () -> (prints nothing)
```

In this example, the metatable we are setting for o is not

contains the __gc metamethod , so the object is not marked for phi-
nalization. Even if we later add the __gc field to the metatable,
Lua does not consider this assignment as special, so
object and will not be marked. As we said, this is rarely a trial.
lemo; usually the metatable does not change after it has been
assigned by metatable.
If you really want to set the metamethod later, then you can
those to use any value for the __gc field as temporary:

```
o = {x = "hi"}
mt = {__gc = true}
setmetatable (o, mt)
mt .__ gc = function (o) print (ox) end
o = nil
collectgarbage () -> hi
```

Now, as metatable field contains the __gc , object o po-
tossed about for finalization. There is no problem in getting
give a metamethod later; Lua only calls the finalizer if it
is a function.
When the garbage collector destroys multiple objects in one
and in the same loop, it calls their finalizers in the reverse order
the one in which the objects were marked for finalization. Consider
Rome the following example that creates a linked list of objects
with finalizers:

```
mt = {__gc = function (o) print (o [1]) end}
list = nil
for i = 1, 3 do
list = setmetatable ({i, link = list}, mt)
end
list = nil
collectgarbage ()
-> 3
-> 2
-> 1
```

The first object to be finalized will be object 3, which was
the last marked object.
A common misconception is that linking
ki between the destroyed objects can affect the order in
where they will be finalized. For example, you might think that
object 2 in the previous example must be finalized before
object 1, since there is a link from 2 to 1. However, the links
can form cycles. Therefore, they do not impose any
order for finalization.

Another subtle point related to finalizers is
recovery . When the finalizer is called, it receives a fi
an object to be parsed as a parameter. So the object
becomes alive again, at least during finalization.
I call this *temporary recovery* . At run time
the finalizer does not prevent it from remembering the object, for example in
a global variable so that the object remains accessible
after the finalizer finishes. I call it *permanent*
restoration .
Recovery must be transitive. Consider the following-
snippet of code:

```
A = {x = “this is A”}
B = {f = A}
setmetatable (B, {__gc = function (o) print (ofx) end})
A, B = nil
collectgarbage () -> this is A
```

The finalizer for B refers to A , so A cannot be removed.
flax before finalizing Bed and . Lua must reconstruct both A and B before
by calling the finalizer.
Due to restoration, objects with finalizers have restored
are carried out in two passes. The garbage collector first finds out that
an object with a finalizer is unreachable (no one refers to it),
then it restores that object and adds it to the queue for
finalization. After executing the finalizer, Lua marks the object
as finalized. The next time the garbage collector is
finds out that the object is unreachable, he will destroy it. If you want to
ensure that all the garbage in your program is truly its own
run, then you must call collectgarbage twice; second call
will destroy objects that were finalized during the first
first call.
The finalizer for each object is executed exactly once,
because Lua marks objects that have already been finalized. If the volume
the object has not been deleted before the end of the program, then Lua will call
it in
the very end. This feature allows you to implement in Lua an analog
atexit functions, that is, functions that are called directly
immediately before exiting the program. Anything for this
what you need is to create a table with a finalizer and remember the link
to it somewhere, for example in a global variable:

```
_G.AA = {__gc = function ()
```

```
- your 'atexit' code comes here
print ("finishing Lua program")
end}
setmetatable (_G.AA, _G.AA)
```

Another interesting possibility is the ability to call
call a specific function every time Lua exits the loop
garbage collection. Since the finalizer is called exactly once,
then you need to create a new object in the finalizer to call the next
blowing finalizer:

```
do
local mt = {__gc = function (o)
- whatever you want to do
print ("new cycle")
- create a new object for the next cycle
setmetatable ({}, getmetatable (o))
end}
- create the first object
setmetatable ({}, mt)
end
collectgarbage () -> new loop
collectgarbage () -> new loop
collectgarbage () -> new loop
```

Interaction of objects with finalizers and weak tables with
keeps a subtle moment. The garbage collector cleans up values in a weak
table before restoring while keys are being flushed
after recovery. The following code snippet illustrates this
behavior:

```
- table with weak keys
wk = setmetatable ({}, {__mode = "k"})
- table with weak values
wv = setmetatable ({}, {__mode = "v"})
o = {} - object
wv [1] = o; wk [o] = 10 - add to both tables
setmetatable (o, {__gc = function (o)
print (wk [o], wv [1])
end})
o = nil; collectgarbage () -> 10 nil
```

During the execution of the finalizer, it finds the object in the table
wk, but not in the wv table. The rationale for this behavior is
that we often store object properties in tables with weak keys
(as we discussed in Section 17.3) and finalizers can be
need to refer to these attributes. However, we use tab-
faces with weak values to reuse existing ones
objects; in this case, the finalizable objects are no longer needed.

Exercises

Exercise 17.1. Write a code to check if
does Lua really use ephemeral tables. (Not for-
call collectgarbage for garbage collection.)
check your code in both Lua 5.1 and Lua 5.2.

Exercise 17.2. Consider the first example from section 17.6,
creates a table with a finalizer that prints the message
on call. What happens if the program ends
without calling garbage collection? What happens if the program is
calling os.exit ? What happens if the program terminates
execution with error?

Exercise 17.3. Suppose you need to implement caching
a table for a function that takes a string and returns
string. Using a weak table will prevent deletions
records as weak tables do not consider rows
as objects to be deleted. How can you implement caching
in this case?

Exercise 17.4. Explain the output of the following program:

```
local count = 0
local mt = {__gc = function () count = count - 1 end}
local a = {}
for i = 1, 10000 do
count = count + 1
a [i] = setmetatable ({}, mt)
end
collectgarbage ()
print (collectgarbage "count" * 1024, count)
a = nil
collectgarbage ()
print (collectgarbage "count" * 1024, count)
collectgarbage ()
print (collectgarbage "count" * 1024, count)
```

Part III
WITH TANDARD LIBRARIES

Chapter 18
Mathematical library

In this and the following chapters on the standard library, my goal is not to give a complete specification of each function tion, but to show what functionality each library. I can omit some specific options, or behavior for clarity. The main goal is to ignite your curiosity, which can then be satisfied by reading- See the Lua documentation.

The math library contains a standard set of mathematical functions such as trigonometric (sin , cos , tan , asin , acos etc.), exponentiation and logarithm (exp , log , log10), rounding (floor , ceil), min , max , functions for generating pseudo-tea numbers (random , randomseed) and the variables pi and huge (last it is the largest representable number, on some platforms forms can take the special value *inf*).

All trigonometric functions work with radians. You can you can use deg and rad functions to convert between degrees and radians. If you want to work with degrees, you can re-define trigonometric functions:

```
do
local sin, asin, ... = math.sin, math.asin, ...
local deg, rad = math.deg, math.rad
math.sin = function (x) return sin (rad (x)) end
math.asin = function (x) return deg (asin (x)) end
...
end
```

The math.random function generates pseudo-random numbers. we we can call it in three different ways. When we call her with no arguments, it returns a real pseudo-random number in the range [0, 1). When we call it with a single argument, integer *n* , then it returns a pseudo-random integer *x* lying between 1 and *n* . Finally, we can call it with two integer-with arguments *l* and *u* , then it will return a pseudo-random integer, lying between *l* and *u* .

You can set the seed for the pseudo-random generator

numbers using the randomseed function ; her only clean
the primary argument is "seed". Usually when starting work
programs the pseudo-random number generator is initialized
some fixed value. This means that every time
when you run your program it generates the same
a sequence of pseudo-random numbers. For debugging, it turns out
is very useful, but in the game you will always get one and
also. The standard trick to combat this is to use
setting the current time as a "seed" using a call
math.randomseed (os.time ()) . The os.time function returns a number,
representing the current time, usually as a number of seconds,
marching from a certain date.
The math.random function uses the rand function from the standard
libraries of the C language. In some implementations, returns numbers with not
very good statistical properties. You can reverse-
to independent distributions in search of a better generator
pseudo-random numbers. (Lua standard distribution does not include
into itself a similar generator due to copyright issues. It
contains only code written by the Lua authors.)

Exercises

Exercise 18.1. Write a function to check if
is the given number a power of two.
Exercise 18.2. Write a function to calculate the volume of the cone
sa by its height and angle between its generatrix and axis.
Exercise 18.3. Implement another pseudo-random generator
numbers for Lua. Search the internet for a good algorithm. (You
a library for bitwise operations may be needed.)
Exercise 18.4. Using the math.random function , write
function to get pseudo-random numbers from Gaussian
distribution.
Exercise 18.5. Write a function for mixing
of this list. Make sure all options are equally likely.

Chapter 19

Library for bitwise operations

A constant source of complaints about Lua is the lack of
bitwise operations. This absence is by no means accidental. Not so easy
reconcile bitwise operations with floating point numbers.
We can express some of the bitwise operations as arithmetic
tic operations. For example, shifts to the left correspond to multiplication
by a power of two, shifts to the right correspond to division. One-
bitwise AND and OR have no such arithmetic counterparts. They
defined for binary representations of integers. Practically
it is not possible to extend them to floating point operations. Even
some simple operations are meaningless. What should be additional
by 0.0? Should it be -1? Or 0xFFFFFFFF (what's in
Lua is 4,294,967,295, which is clearly not -1)? Or maybe $2^{64} - 1$
(a number that cannot be accurately represented using a value like
double)?
To avoid such problems, Lua 5.2 introduces
operations using a library, not as built into the language
operations. This makes it clear that these operations are not "genus"
ny "for numbers in Lua, but they use a certain interpretation.
for working with these numbers. Moreover, other libraries
may suggest other interpretations of bitwise operations (e.g.
measures using more than 32 bits).
For most of the examples in this chapter, I will use six
decimal notation. I will use the word MAX to indicate
values 0xFFFFFFFF (that is, $2^{32} - 1$). In the examples I will use
the following additional function:

```
function printx (x)
print (string.format (“0x% X”, x))
end
```

The bitwise library in Lua 5.2 is called bit32 . As follows from
name, it works with 32-bit numbers. Since **and** , **or** and **not**
are reserved words in Lua, then the corresponding
the functions are named band , bor and bnot . For the sequence in the name
The function for bitwise exclusive OR is named bxor :

```
printx (bit32.band (0xDF, 0xFD)) -> 0xDD
printx (bit32.bor (0xD0, 0x0D)) -> 0xDD
printx (bit32.bxor (0xD0, 0xFF)) -> 0x2F
printx (bit32.bnot (0))
-> 0xFFFFFFFF
```

The band , bor and bxor functions accept any number of arguments.
Tov:

```
printx (bit32.bor (0xA, 0xA0, 0xA00)) -> 0xAAA
printx (bit32.band (0xFFA, 0xFAF, 0xAFF)) -> 0xAAA
printx (bit32.bxor (0, 0xAAA, 0))
-> 0xAAA
printx (bit32.bor ())
-> 0x0
printx (bit32.band ())
-> 0xFFFFFFFF
printx (bit32.bxor ())
-> 0x0
```

(They are all commutative and associative.)
The bitwise library works with unsigned integers.
During operation, any number passed as an argument is converted to an integer in the range 0-MAX . First, the unspecified numbers are ok-swear in an unspecified way. Second, numbers out of range 0-MAX are converted to it using the modulus operation: integer *n* becomes *n* % (2 32). This operation is equivalent to getting the binary representation of the number and then taking its least significant 32 bits.
As expected, -1 becomes MAX . You can use the following-operations to normalize a number (that is, to display it in range 0-MAX):

```
printx (bit32.bor (2 ^ 32))
-> 0x0
printx (bit32.band (-1))
-> 0xFFFFFFFF
```

Of course, in standard Lua it's easier to just do n% (2 ^ 32) .
Unless explicitly specified, all functions in the library return re-the result, which also lies in 0-MAX . However, you should be tricky when using the results of bitwise operations in as ordinary numbers. Sometimes Lua is compiled using other second type for numbers. In particular, some systems with limited capabilities use 32-bit numbers as numbers in Lua.
In these systems, MAX = -1 . Moreover, some bitwise libraries thecas use different conventions for their results. By-this whenever you need to use the result of the bitwise operations as a number, be careful. Avoid comparisons: instead of x <0 write bit32.btest (x, 0x80000000) . (We will soon see dim the btest function .) Use the bitwise library itself to normalizing constants:

```
if bit32.or (a, b) == bit32.or (-1) then
<some code>
```

The bitwise library also defines operations for shifting and bit rotation: lshift to shift left; rshift and arshift to shift-ha to the right; lrotate for left rotation and rrotate for rotation right. Except for arithmetic shift (arshift), all shifts fill new bits with zeros. Arithmetic shift for fills the bits on the left with copies of its last bit.

```
printx (bit32.rshift (0xDF, 4))
-> 0xD
printx (bit32.lshift (0xDF, 4))
-> 0xDF0
printx (bit32.rshift (-1, 28))
-> 0xF
printx (bit32.arshift (-1, 28))
-> 0xFFFFFFFF
printx (bit32.lrotate (0xABCDEF01, 4)) -> 0xBCDEF01A
printx (bit32.rrotate (0xABCDEF01, 4)) -> 0x1ABCDEF0
```

Shift or rotation by a negative number of bits shifts (rotate em) in the opposite direction. For example, shift -1 bit to the right is equivalent to shifting 1 bit to the left. The result of a shift by more than 31 bits is 0 or MAX because all original bits are gone:

```
printx (bit32.lrotate (0xABCDEF01, -4)) -> 0x1ABCDEF0
printx (bit32.lrotate (0xABCDEF01, -36)) -> 0x1ABCDEF0
printx (bit32.lshift (0xABCDEF01, -36)) -> 0x0
printx (bit32.rshift (-1, 34))
-> 0x0
printx (bit32.arshift (-1, 34))
-> 0xFFFFFFFF
```

In addition to these more or less standard operations, bitwise the library also provides three additional functions.

The btest function performs the same operation as band , but returns Returns the result of comparing a bitwise operation with zero:

```
print (bit32.btest (12, 1))
-> false
print (bit32.btest (13, 1))
-> true
```

Another common operation is to retrieve specified bits from the number. Usually this operation involves shifting and bitwise AND; a bitwise library packs it all into one function. The call bit32.extract (x, f, w) returns w bits from x , on-starting from bit f :

```
printx (bit32.extract (0xABCDEF01, 4, 8)) -> 0xF0
```

```
printx (bit32.extract (0xABCDEF01, 20, 12)) -> 0xABC
printx (bit32.extract (0xABCDEF01, 0, 12)) -> 0xF01
```
This operation counts bits from 0 to 31. If the third argument is
ment (w) is not specified, then it is considered equal to one:
```
printx (bit32.extract (0x0000000F, 0)) -> 0x1
printx (bit32.extract (0xF0000000, 31)) -> 0x1
```
The reverse of the extract operation is the replace operation , which
paradise replaces the given bits. The first parameter is the outcome
number. The second parameter specifies the value to be inserted.
The last two parameters, f and w , have the same meaning as in bit32.
extract :
```
printx (bit32.replace (0xABCDEF01, 0x55, 4, 8)) -> 0xABCDE551
printx (bit32.replace (0xABCDEF01, 0x0, 4, 8)) -> 0xABCDE001
```
Note that for any valid values of x , f and w
the following equality holds:
```
assert (bit32.replace (x, bit32.extract (x, f, w), f, w) == x)
```

Exercises

Exercise 19.1. Write a function to check what is
this number is a power of two.
Exercise 19.2. Write a function to calculate a number
single bits in binary representation of a number.
Exercise 19.3. Write a function to check if
Whether the binary representation of a number is a palindrome.
Exercise 19.4. Define shift operations and bitwise
AND using Lua arithmetic operations.
Exercise 19.5. Write a function that receives a string,
encoded in UTF-8, and returns its first character as
number. The function should return nil if the line does not start
with a valid UTF-8 sequence.

Chapter 20
Library
for working with tables

The table library contains additional functions that allow
for working with tables as arrays. It provides
functions for inserting and removing items from the list, for sorting
ki array elements and to concatenate all strings in the array.

20.1. Insert and remove functions

The table.insert function inserts an element at a given location in the array,
shifting the rest of the elements in order to make space. On-
example, if t is an array {10, 20, 30} , then after calling table.
insert (t, 1,15) t will be {15, 10, 20, 30} . Special (and
quite common), the case is to call insert without specifying a
insertion, then the element is inserted at the very end of the array and shift
no elements happen. As an example, the following code reads
input line by line, remembering all lines in the array:

```
t = {}
for line in io.lines () do
table.insert (t, line)
end
print (#t) -> (number of lines read)
```

In Lua 5.0, this technique is fairly common. In later
versions, I prefer to use t [# t + 1] = line in order to
add a string to the array.

The table.remove function removes (and returns) an element from the given
place of the array, shifting the next elements of the array.
If, when calling, the position inside the array was not specified, then delete

The last element of the array is taken.
With these two functions it is quite easy to implement
stacks, queues and double queues. We can initialize
similar structures like t = {} . The operation of adding an item
is equivalent to table.insert (t, x) ; the operation of removing an element of
equivalent
valence table.remove (t). The call to table.insert (t, 1, x) adds
element to the other end of the corresponding structure, and a call to table.
remove (t, 1) accordingly removes an element from that end. Two
the latter operations are not particularly effective as they should
move all elements of the array in memory. However, since the Bible
In the table library, these functions are implemented in C, they are not
are too expensive and work well for small arrays (up to
several hundred elements).

20.2. Sorting

Another useful function for working with arrays is table.
sort ; we've seen it before. It takes as arguments
an array and optionally a function for comparison. This function is
takes two arguments as input and must return ***true*** if the first
the element must come before the second. If this function is not specified, then
the sort function uses the standard '<' operator.
Typical confusion occurs when a programmer tries to
sort the indexes in the table. In the table, the indices form many
property in which there is no ordering. If you want them
sort, then you need to copy them into an array and sort
this array. Let's take an example. May you read the input
file and built a table, which for each function name co-
holds the line where this function was defined: something like
next:

```
lines = {
luaH_set = 10,
luaH_get = 24,
luaH_present = 48,
}
```

And now you need to print these functions in alphabetical order
row. If you traverse this table with the pairs function ,
then the names will appear in no particular order. You can't explicitly

sort as these names are table keys.
However, if you put them in an array, then already this array
can be sorted. So you first need to create an array
with these names, then sort it and only then print
result:

```
a = {}
for n in pairs (lines) do a [#a + 1] = n end
table.sort (a)
for _, n in ipairs (a) do print (n) end
```

Some are confused. After all, Lua arrays don't contain any
what ordering (arrays are actually tables). Poet-
to that we impose ordering when working with indexes that
can be ordered. This is why you are better off traversing the array when
help ipairs , not pairs . The first of these functions sets
the order of the keys is 1, 2, 3, ..., while the second just uses
arbitrary order from the table.
As a more advanced solution, we can write an iteration
A generator for traversing a table using a given key order.
The optional f parameter specifies this order. This sleep iterator is
chala sorts the keys into a separate array, and then bypasses this
array. At each step, it returns a key and the corresponding value
reading from the original array:

```
function pairsByKeys (t, f)
local a = {}
for n in pairs (t) do a [#a + 1] = n end
table.sort (a, f)
local i = 0
return function () - iterating function
i = i + 1
return a [i], t [a [i]]
end
end
```

With this iterator it is easy to print function names in
alphabetical order:

```
for name, line in pairsByKeys (lines) do
print (name, line)
end
```

20.3. Concatenation

We have already seen the table.concat function in section 11.6 . She takes on
input is a list of strings and returns the result of the concatenation of all these

lines. The optional second argument specifies the delimiter string.
There are also two more optional arguments that specify the
dexes of the first and last strings to be concatenated.
The following function is an interesting generalization of table.
concat . It can accept nested lists of strings as input:

```
function rconcat (l)
if type (l) ~ = “table” then return l end
local res = {}
for i = 1, #l do
res [i] = rconcat (l [i])
end
return table.concat (res)
end
```

For each item in the list, rconcat recursively calls itself
for processing nested lists. Then it calls table.
concat to combine intermediate results.

```
print (rconcat {{“a”, {“nice”}}, “and”, {{“long”}, {“list”}}})
-> a nice and long list
```

Exercises

Exercise 20.1. Rewrite the rconcat function so that for
it could be given a separator string:

```
print (rconcat ({{{“a”, “b”}, {“c”}}, “d”, {}, {“e”}}, “;”)
-> a; b; c; d; e
```

Exercise 20.2. The problem with table.sort is that this
sort is not stable (stable sort), that is, ele-
the cops that the sorting function considers to be equal can
They can change their order during the sorting process. How can you
implement robust sorting in Lua?

Exercise 20.3. Write a function to check if
Whether the specified table is a valid sequence.

CHAPTER 21
Library for working with strings

Immediate possibilities of working with strings interpreted
Lua's are pretty limited. The program can create strings, co-
combine them and get the length of the string. But she cannot extract sub-
strings or examine their contents. True power to work
with strings comes from her library for working with strings.
A library for working with strings is available as the string module .
Since Lua 5.1, functions are also exported as string methods
(using metatables). So, a line break in capital letters
can be written as string.upper (s) or s: upper () . Choose
yourself.

21.1. Main functions for working with strings

Some functions for working with strings in the library are extremely
are simple: calling string.len (s) returns the length of string s . She ek-
is equivalent to #s . Calling string.rep (s, n) (or s: rep (n)) returns
string s repeated n times. You can create a 1MB string (on-
example, for tests) with string.rep (“a”, 2 ^ 20) . Call
string.lower (s) returns a copy of the uppercase string-
mi replaced by lowercase; all other characters remain unchanged.
(The string.upper function converts lowercase letters to uppercase.)
As an example, if you want to sort strings outside of the
from uppercase / lowercase letters, you can use the following
next piece of code:

```
table.sort (a, function (a, b)
return a: lower () <b: lower ()
end)
```

The call to string.sub (s, i, j) returns the substring of s starting at
i -th character and ending with j -th (inclusive). In Lua, the first character is
row has index 1. You can also use negative
indices that are counted from the end of the line: index -1 referen-
goes to the last character of the string, –2 to the penultimate character, and so on.
So calling string.sub (s, 1, j) (or s: sub (1, j)) will return
specifies the beginning of a string of length j ; string.sub (s, j, -1) (or just
s: sub (j) because the default for the last argument is
minta is –1) returns the end of the string starting from the j -th character;
and string.sub (s, 2, -2) returns a copy of the string s , in which the
we are the first and last characters:

```
s = "[in brackets]"
```

print (s: sub (2, -2)) -> in brackets
Remember that strings in Lua are immutable. String.sub function ,
like any other function in Lua, it doesn't change the value of the string, but
returns a new string. A common mistake is to use
something like s: sub (2, -2) and expect this to change the value of st-
ki s . If you want to change the value of a variable, then you must
assign a new value to it:
s = s: sub (2, -2)
The string.char and string.byte functions translate between characters
and their internal numeric representations. String function
char takes integers as input, converts each of them to sym-
ox and returns a string built from all these characters. Call
string.byte (s, i) returns internal numeric representation
i -th character of string s ; the second argument is optional, the call to string.
byte (s) returns the internal numeric representation of the first
string character s . In the following examples, we assume that the characters
represented by ASCII encoding:
print (string.char (97))
-> a
i = 99; print (string.char (i, i + 1, i + 2)) -> cde
print (string.byte ("abc"))
-> 97
print (string.byte ("abc", 2))
-> 98
print (string.byte ("abc", -1))
-> 99
On the last line, we used a negative index for
access to the last character of the line.
Since Lua 5.1, the string.byte function supports the third one,
optional argument. Calling string.byte (s, i, j) returns
numerical representations of all characters at once between
indices i and j (inclusive):
print (string.byte ("abc", 1, 2)) -> 97 98
The default for j is i , so the call without
the third of its argument returns the i- th character. Calling {s: byte (1, -1)}
creates a table with codes of all characters in string s . According to this table
we can get the original string by calling string.
char (table.unpack (t)) . This trick does not work for very long
lines (more than 1 MB), since Lua has a limit on the number of
values rotated by the function.
The string.format function is a powerful tool for

formatting strings, usually for output. She returns the
a matted version of its arguments (supported by
free number of arguments) using the description given by its first
argument, the so-called *format string* . For this line
there are rules similar to those for the printf function from
the standard C library: it consists of plain text and
pointers that control where and how to place each argument
cop in the resulting string. The pointer consists of a '%' character ,
followed by a character specifying how to format the argument
ment: 'd' for decimal numbers, 'x' for hexadecimal numbers,
'o' for octal, 'f' for floating point numbers, 's' for
lines, there are also some other options. Between '%' and sym-
there may be other options that specify formatting,
such as the number of decimal digits for a floating point number:

```
print (string.format (“pi =% .4f”, math.pi)) -> pi = 3.1416
d = 5; m = 11; y = 1990
print (string.format (“% 02d /% 02d /% 04d”, d, m, y)) -> 05/11/1990
tag, title = “h1”, “a title”
print (string.format (“<% s>% s </% s>”, tag, title, tag))
-> <h1> a title </h1>
```

In the first example, % .4f is a floating point number with
three digits after the decimal point. In the second example, % 02d
denotes a decimal number of at least two digits, if necessary
walkability padded with zeros; % 2d without zero will pad the number
spaces. For a complete description of these options, refer to the
See the Lua manual, or refer to the C manual.
since Lua uses the C library to do all
hard work here.

21.2. Functions for work with templates

The most powerful functions in the library for working with strings are
mi are find , match and gsub (global substitution) functions
and gmatch (global search). They are all *template* based .
Unlike a number of other scripting languages, Lua does not use

neither POSIX syntax nor language syntax for working with templates
Perl. The main reason for this decision is size: typical
ny implementation of POSIX regular expressions takes over 4000
lines of code. This is larger than all the standard Lua libraries, taking
together. For comparison, the implementation of working with templates in Lua
is
takes less than 600 lines. Of course, the implementation of working with
templates in
Lua is inferior to a full-fledged POSIX implementation. Still working with
templates in Lua is a powerful tool and includes
some features that are difficult to relate to standard
POSIX implementations.

String.find function

The string.find function searches for a given pattern within a string. Pros-
The strongest case of a pattern is a word that matches
your copy. For example, the pattern 'hello' will search for the substring
“Hello” within the entire given string. When finding template find
returns two values: the index from which the co-
drop, and the index where the match ends. If the match is not
found, then ***nil is*** returned :

```
s = “hello world”
i, j = string.find (s, “hello”)
print (i, j)
-> 1 5
print (string.sub (s, i, j))
-> hello
print (string.find (s, “world”)) -> 7 11
i, j = string.find (s, “l”)
print (i, j)
-> 3 3
print (string.find (s, “lll”))
-> nil
```

When the search for the template is completed successfully, we can call
string.sub with returned values in order to get
part of the original string that matches the pattern. For simple
templates, such a string will be the template itself.
The string.find function has an optional third parameter: in-
dex specifying where within the string the search should start.
This parameter is useful when we want to get all
pattern occurrences: in this case we call the search function
repeatedly, each time starting the search after the position at which

a previous match was found. As an example, the following
the following code builds a table with the positions of all '\ n' characters inside lines:

```
local t = {}
- table for storing indexes
local i = 0
while true do
i = string.find (s, “\ n”, i + 1) - looking for the next occurrence
if i == nil then break end
t [#t + 1] = i
end
```

Later we will see an easier way to write such loops,
using the iterator string.gmatch .

String.match function

The string.match function is similar to string.find in the sense that it
also searches for occurrences of a pattern in a string. However, instead of
return the position where the pattern was found, it returns the part
lines matching the pattern:

```
print (string.match (“hello world”, “hello”)) -> hello
```

For simple templates like 'hello' this function is meaningless.
la. It shows its power when used with difficult
templates as in the following example:

```
date = “Today is 17/7/1990”
d = string.match (date, “% d + /% d + /% d +”)
print (d) -> 7/17/1990
```

We will discuss shortly both the meaning of the pattern '% d + /% d + /% d +' and more complex use of string.match .

String.gsub function

The string.gsub function has three required parameters:

ku, pattern and replacement string. She is used to replace all
occurrences of the pattern in the original string by the given string:

```
s = string.gsub (“Lua is cute”, “cute”, “great”)
print (s)
-> Lua is great
s = string.gsub (“all lii”, “l”, “x”)
print (s)
-> axx xii
s = string.gsub (“Lua is great”, “Sol”, “Sun”)
print (s)
-> Lua is great
```

The optional fourth parameter limits the number of
replaceable replacements:

```
s = string.gsub (“all lii”, “l”, “x”, 1)
print (s)
-> axl lii
s = string.gsub (“all lii”, “l”, “x”, 2)
print (s)
-> axx lii
```

The string.gsub function also returns as the second value
the number of replacements performed. For example, in a simple way,
th the number of spaces in a line is

```
count = select (2, string.gsub (str, ““, ““))
```

String.gmatch function

The string.gmatch function returns a function that iterates over
all occurrences of the pattern in a string. For example, the following example co-
takes all words in a given string s :

```
words = {}
for w in string.gmatch (s, “% a +”) do
words [#words + 1] = w
end
```

As we will discuss shortly, the pattern '% a +' matches the occurrence
one or more letters (i.e. words). Therefore, the cycle is
denotes all the words within the string, storing them in the words table .
The following example implements a function similar to package.
searchpath using gmatch and gsub :

```
function search (modname, path)
modname = string.gsub (modname, “%.”, “/”)
for c in string.gmatch (path, “[^;] +”) do
local fname = string.gsub (c, “?”, modname)
local f = io.open (fname)
if f then
f: close ()
return fname
```

```
end
end
return nil - not found
end
```

The first step is to replace all dots with a delimiter in the path, which is considered equal to '\' . (As we will see below, the point has special meaning in templates. To match the point we should write '%.'). Then the function iterates over all the components paths where all questions are replaced for each component adjectives to the module name and it is checked if such what file. If so, the function closes this file and returns it name.

21.3. Templates

You can make templates more useful with *classes characters* . A character class is an element in a template that can match any character from the given set. For example measures, class % d matches any digit. Therefore, one can search for date in dd / mm / yyyy format using template % d% d /% d% d / % d% d% d% d :

```
s = “Deadline is 30/05/1999, firm”
date = "% d% d /% d% d /% d% d% d% d"
print (string.sub (s, string.find (s, date))) -> 30/05/1999
```

The following table contains a list of all character classes:

...

All symbols

% a

Letters

% c

Control characters

% d

Numbers

% g

Printed characters other than space

% l

Lower case

% p

Punctuation symbols

% s

Whitespace characters

% u

Lower case

% w
Letters and numbers
% x
Hexadecimal digits
If you use the appropriate class as the class name
capital letter, then it corresponds to the complement of the class (that is, all
characters outside the class). For example, '% A' matches all
non-letters:
print (string.gsub (“hello, up-down!”, “% A”, “.”))
-> hello..up.down. 4
(4 are not part of the resulting string. This is the second value.
The value returned by gsub is the total number of replacements performed. I will
further omit this number in the following examples that print the results
tat calling gsub .)
Some symbols, called *magic symbols* , have
a special value within a template. Magic symbols are
go
(). % + - *? [] ^ $
The '%' character is used to insert these characters into the pattern.
So, '%.' corresponds to the point; '%%' matches the character itself
'%' . You can use '%' like this not only with ma-
symbols, but also with any non-alphanumeric symbols
oxen. When in doubt, use '%' instead .
For a Lua parser, templates are just plain strings. They
obey the same rules as the rest of the lines. Only
functions for working with templates treat them as templates,
and only these functions use the special character meaning
'%' . To put quotes inside a template, use the same
the very tricks used to put quotes inside
other lines.
You can also create your own classes by grouping different
personal classes and symbols within square brackets. For instance,
class '[% w_]' matches alphanumeric characters and sym-
ox underlining; class '[01]' matches binary digits;
class '[% [%]]' matches square brackets. In order to
count the number of vowels in the text, you can use the following-
the following code:
nvow = select (2, string.gsub (text, “[AEIOUaeiou]”, “”))
You can also include ranges of sym-
oxen, writing down the first and last characters, separated by signs

lump minus. I rarely use this, as most of the
the ranges used are already defined; for example, '[0-9]' is
is the same as '% d' and '[0-9a-fA-F]' is the same as
'% x' . However, if you need octal digits, then you can
you can use '[0-7]' instead of '[01234567]' , you can also
get the complement of any class by prefixing the '^' character :
so the pattern '[^ 0-7]' matches any character that is not
an octal digit, and '[^ \ n]' matches any character from-
personal from '\ n' . However, remember that built-in classes can be
before it is easier to use the capitalized variant: '% S' is easier than
'[^% s]' .
Templates can be made more useful by using the mod-
fixers for specifying the number of repetitions and optional parts.
Lua templates offer four such modifiers:
+
1 or more reps
*
0 or more reps
-
0 or shorter reps
?
Optional (0 or 1 time)
The modifier '+' matches one or more sim-
oxen of a given class. It will always return the longest occurrence
template. For example, the pattern '% a +' denotes one or more letters,
that is the word:
print (string.gsub (“one, and two; and three”, “% a +”, “word”))
-> word, word word; word word
The pattern '% d +' matches one or more digits (meaning
an unsigned number):
print (string.match (“the number 1298 is even”, “% d +”)) -> 1298
The modifier '*' is similar to '+' , but it also allows null
the number of occurrences of characters from this class. Used frequently
to indicate optional spaces between template parts.
For example, for a pattern matching a pair of parentheses (perhaps
with spaces in between), you can use the following pattern:
'% (% s *%)' : pattern '% s *' matches zero or more
spaces between brackets. (The parentheses also have a special meaning
patterns, so we specify them using the '%' character .)
As another example, the pattern '[_% a] [_% w] *' matches
identifiers inside a Lua program: starts with a space

or underscore followed by zero or more
underscores and alphanumeric characters.
Like '*' , the modifier '-' also matches zero or
more characters of the given class. However, instead of
matches the longest sequence it matches
the shortest sequence. Sometimes there is nothing in between
what a difference, but usually they give different results. For instance,
if you try to find an id using a template
'[_% a] [_% a] -' , then you will receive only the first character of the identification
torus because '[_% a] -' matches an empty sequence.
On the other hand, let's say you want to find comments in a program
me in C. Most will try to use the pattern '/%*.*%*/'
(that is, "/ *" followed by any sequence of characters
fishing followed by "* /"). However, since '. *' Will try
match as many characters as possible, then the first "/ *"
will close only with the most recent "* /" in the program:

```
test = "int x; / * x * / int y; / * y * / "
print (string.match (test, "/%*.*%*/"))
-> / * x * / int y; / * y * /
```

The pattern '.-' captures the least number of characters required
given for the first "* /", and thus gives the desired result:

```
test = "int x; / * x * / int y; / * y * / "
print (string.gsub (test, "/%*.-%*/", ""))
-> int x; int y;
```

Last modifier '?' matches optional with-
accompanying symbol. For example, let's say we want to find the number
in text, which can contain an optional character. Template
'[+ -]?% d +' successfully copes with the job, finding such
numbers like "-12" , "23" and "+1009" . The class '[+ -]' matches
either the '+' character or the '-' character ; the next character '?' de-
barks this character optional.
Unlike other systems, in Lua the modifier can be
changed to character class only; you cannot group templates under
one modifier sign. For example, there is no template matching
an optional word (unless it consists of one
symbol). Usually this limitation can be bypassed by
advanced techniques that we will see at the end of this chapter.
If the pattern starts with a '^' character then it will match-
only with the beginning of the line. Likewise if the pattern ends
with the '$' character , it will match only the end of the string. You

you can use both of these symbols to create templates. On-
example, the following test checks if a string starts with a digit:
```
if string.find (s, “^% d”) then ...
```
The following test verifies that the string is a number, with no other
their characters at the beginning or end:
```
if string.find (s, “^ [+ -]?% d + $”) then ...
```
The characters '^' and '$' have this meaning only when they come across
are located at the beginning or at the end of the line, respectively. Otherwise, they will
fall like ordinary symbols.
Another element in the template is '% b' . We write it down
like '% b *xy* ' where *x* and *y* are two different characters; character *x will* speak
falls as the opening character and *y* as the closing character . For instance,
the pattern '% b ()' matches the part of the string that starts with '('
and ends with ')' :
```
s = “a (enclosed (in) parentheses) line”
print (string.gsub (s, “% b ()”, “”)) -> a line
```
We usually use this pattern as '% b ()' , '% b []' , '% b {}'
or '% b <>' , but you can use any
bye symbols.
Finally, the '% f [*char-set*]' element is a *border pattern* . is he
defines the place where the next character is contained in the class
char-set and the previous one is not:
```
s = “the anthem is the theme”
print (s: gsub (“% f [% w] the% f [% W]”, “one”))
-> one anthem is one theme
```
The pattern '% f [% w]' matches the border between non-alphabetic
numeric and alphanumeric characters, and the pattern '% f [% W]'
matches the boundary between an alphanumeric character and a non-
alphanumeric character. Therefore, the given template corresponds
matches the string "the" as a whole word. note that
we must write many symbols inside square
side even when it's just one class.
Positions before the first and after the last are interpreted as
holding the character with code 0. In the previous example, the first "the"
starts with a border between a null character (not in class ' [% w]')
and 't' (in class '[% w]').
The border pattern was implemented in Lua 5.1, but is not documented.
It only became official in Lua 5.2.

21.4. Grips

The *capture* mechanism allows the template to remember parts of the string, satisfying
elements of the template for later use. You
you can specify the capture by capturing the parts of the template you want to capture
enough, inside parentheses.
When there are captures in the template, the string.match function will return
Spreads each captured value as a separate result; others
in words, it splits the string into its captured pieces.

```
pair = “name = Anna”
key, value = string.match (pair, “(% a +)% s * =% s * (% a +)”)
print (key, value) -> name Anna
```

The pattern '% a +' specifies a non-empty sequence of letters; template
'% s *' specifies a possibly empty sequence of spaces. By-
for this in the example above, the entire pattern defines a sequence of letters,
followed by a sequence of spaces, followed by
there is an equal sign '=' , followed again by the sequence
spaces followed by another sequence of letters. Both
their sequences of letters, their corresponding patterns are
are in parentheses, so they will be captured on match.
A similar example follows:

```
date = “Today is 17/7/1990”
d, m, y = string.match (date, “(% d +) / (% d +) / (% d +)”)
print (d, m, y) -> 17 7 1990
```

Inside the template, element '% *d* ' , where *d* is a digit, matches a copy
d -th captured line. As an example, consider the case when
yes, you want to find a substring enclosed in ordinary ones inside a string
or double quotes. You can try the pattern '[“'] .- [“ ']' ,
that is, a quotation mark followed by anything followed by
another quote; but you will have problems with lines like
“It's all right” . To solve this problem, we can capture
the first quote and use that to specify the second quote:

```
s = [[then he said: “it's all right”!]]
q, quotedPart = string.match (s, “([\” ']) (.-)% 1 ”)
print (quotedPart)
-> it's all right
```

print (q)
-> “
The first captured value is the quote character itself, and the second
the captured value is a substring between quotes (substring,
satisfying '.-').
For another similar example, we can take a template with
corresponding to long lines in Lua:
% [(= *)% [(.-)%]% 1%]
It matches an opening square bracket followed by
followed by zero or more equal signs, followed by
blows another open square bracket followed by that
anything (line itself) followed by a closing square
a parenthesis followed by the same number of equal signs,
followed by another closing square bracket:
p = “% [(= *)% [(.-)%]% 1%]”
s = “a = [= [[[something]]] ==]] =]; print (a) "
print (string.match (s, p)) -> = [[something]]] ==]
The first capture is a sequence of equal signs (in the example
re only one character); the second captured value is the string itself.
Also the captured values can be used in the override
line in gsub . Like the template, the replacement string can contain
the elements '% *d* ' , which are replaced with the corresponding captured
values when performing substitution. In particular, the element
'% 0' matches the entire portion of the string that matches the pattern.
(Note that the '%' character in the replacement string must be written
sounded like '%%' .) Another example:
print (string.gsub (“hello Lua!”, “% a”, “% 0-% 0”))
-> h-he-el-ll-lo-o L-Lu-ua-a!
The following example rearranges adjacent characters:
print (string.gsub (“hello Lua”, “(.) (.)”, “% 2% 1”)) -> ehll ouLa
As a more useful example, let's write a simple pre-
format builder that receives a string with commands as input
in LaTeX style and translates them into XML format:
\ command {some text} -> <command> some text </command>
If we prohibit nested commands, then the next call
string.gsub does the job:
s = [[the \ quote {task} is to \ em {change} that.]]
s = string.gsub (s, “\\ (% a +) {(.-)}”, “<% 1>% 2 </% 1>”)
print (s)
-> the <quote> task </quote> is to change that.
(We'll see how to handle nested commands later.)
Another useful example is removing spaces from the beginning.

la and end of line:

```
function trim (s)
return (string.gsub (s, “^% s * (.-)% s * $”, “% 1”))
end
```

Pay attention to the careful use of formats. Two anchors ('^' and '$') ensure that we get the entire string. Since-ku '.-' tries to pick the shortest string, then two patterns '% s *' captures all white space around the edges. Also note that since gsub returns two values, we use a circle-left brackets to discard excess (number of substitutions).

21.5. Substitutions

Instead of a string as the third argument to string.gsub, we can use a function or table. When using the function string.gsub calls the function every time it finds a substring matching the pattern; arguments of each call are the captured values are captured, and the function's return value used as a replacement string. When the third argument is is a table, the string.gsub function turns into a table face using the first captured value as a key and the resulting the value from the table as a replacement string. If received from function or table value is ***nil*** , then for a given match no replacement is made.

As a first example, consider the execution of a simple settings - each occurrence of $ varName is replaced with the value of the glo-point variable varName :

```
function expand (s)
return (string.gsub (s, “$ (% w +)”, _G))
end
name = “Lua”; status = "great"
print (expand (“$ name is $ status, isn't it?”))
-> Lua is great, isn't it?
```

For each match with the pattern '$ (% w +)' (dollar sign, for followed by a variable name) the gsub function looks for a matching variable in _G , the found value replaces the occurrence of the pattern bosom into a string. When there is no corresponding variable in the table, then replacement is not made:

```
print (expand (“$ othername is $ status, isn't it?”))
-> $ othername is great, isn't it?
```

If you are not sure if the corresponding variables have
string values, then you can try tostring
to these values. In this case, as a substitute value, you
you can use the function:

```
function expand (s)
return (string.gsub (s, "$ (% w +)", function (n)
return tostring (_G [n])
end))
end
print (expand ("print = $ print; a = $ a"))
-> print = function: 0x8050ce0; a = nil
```

Now, for each match against the pattern '$ (% w +)' gsub is called
Gets the specified function, passing the name as an argument; function of
rotates the replacement value.
In the last example, we go back to the format conversion
teams. We again want to transform commands from LaTeX style
(\ example {text}) to XML style (<example> text </example>), but on
this time we will process nested commands. Next
the function uses recursion to solve our problem:

```
function toxml (s)
s = string.gsub (s, "\\ (% a +) (% b {})", function (tag, body)
body = string.sub (body, 2, -2) - remove brackets
body = toxml (body) - processing nested commands
return string.format ("<% s>% s </% s>", tag, body, tag)
end)
return s
end
print (toxml ("\\ title {The \\ bold {big} example}"))
-> <title> The <bold> big </bold> example </title>
```

URL encoding

For our next example, we will use the *coding*
The URL that HTTP uses to pass parameters in the URL.
This encoding replaces special characters (such as '=' , '&'
and '+') to '% *xx* ' , where *xx* is the character hex code. After
it then replaces spaces with '+' . For example, the string "a + b = c" Bu
det is coded as "a% 2Bb +% 3D + c" . Also the name of each parameter
and its value with an equal sign between them is added to the total
On the next line, the variables are separated from each other by '&' .
For example, the values

name = “al”; query = “a + b = c”; q = "yes or no"
will be encoded as “name = al & query = a% 2Bb +% 3D + c & q = yes + or + no” .
Now suppose we want to decode such a URL and write each
get the value into the table by its name. The next function is to
completes similar decoding:

```
function unescape (s)
s = string.gsub (s, “+”, ““)
s = string.gsub (s, “%% (% x% x)”, function (h)
return string.char (tonumber (h, 16))
end)
return s
end
```

The first operator replaces every '+' with a space. The second finds
hexadecimal encoded characters and for
each such character calls an anonymous function. This function
ration converts the hexadecimal representation to a number (tonumber
base 16) and returns the corresponding character (string.
char). For instance:

```
print (unescape (“a% 2Bb +% 3D + c”)) -> a + b = c
```

To decode name = value pairs, we use the function
gmatch . Since both name and value cannot contain characters
'&' and '=' , then we can use the pattern '[^ & =] +' :

```
cgi = {}
function decode (s)
for name, value in string.gmatch (s, “([^ & =] +) = ([^ & =] +)”) do
name = unescape (name)
value = unescape (value)
cgi [name] = value
end
end
```

The gmatch function call finds pairs of the form name = value . For every
Doing such a pair, the iterator returns the captured values (output
bracketed in the template) as the values of the name and value fields . Body
the loop just calls unsecape on both of those lines and writes
matching pair into cgi table .
It is also easy to write down the corresponding coding. To start
we will write the escape function ; this function encodes all special
digits like '%' followed by a hexadecimal code
character (for the format function , the "% 02X" option is used ,
fetching a two-digit string), and then replaces spaces
to the '+' symbol :

```
function escape (s)
```

```
s = string.gsub (s, “[& = + %%% c]”, function (c)
return string.format (“%%% 02X”, string.byte (c))
end)
s = string.gsub (s, ““, “+”)
return s
end
```

The encode function traverses the entire table that needs to be encoded.
and builds the resulting string:

```
function encode (t)
local b = {}
for k, v in pairs (t) do
b [#b + 1] = (escape (k) .. “=” .. escape (v))
end
return table.concat (b, “&”)
end
t = {name = “al”, query = “a + b = c”, q = “yes or no”}
print (encode (t)) -> q = yes + or + no & query = a% 2Bb +% 3D + c & name = al
```

Replacing tabs

The empty capture '()' in Lua has a special meaning. Instead of
in order not to capture anything (which is completely unnecessary), this template
lone captures the current position within the string as a number:

```
print (string.match (“hello”, “() ll ()”)) -> 3 5
```

(Note that the result of this example is different from
calling string.find , since the position of the second captured value
reading comes after the found pattern.)
A nice example of using this feature is
replacing tab characters with the appropriate number of spaces:

```
function expandTabs (s, tab)
tab = tab or 8 - tab size (default 8)
local corr = 0
s = string.gsub (s, “() \ t”, function (p)
local sp = tab - (p - 1 + corr)% tab
corr = corr - 1 + sp
return string.rep (““, sp)
end)
return s
end
```

The gsub call finds all tabs within a string,
grabbing their position. For each tab character, the internal
The new function uses this position to compute
the number of spaces it takes to get the position,

multiple of the value of tab : it first subtracts one for
position transfer starting at zero and then adding corr to account
previously encountered tabs (replacing each tab character affects
to the positions of the following characters). Then the correction is calculated
for the next tab character: minus one for the tab to be deleted
plus sp to account for the added spaces. Finally, she returns
a string with the appropriate number of spaces.
For completeness, let's look at how you can reverse this operation.
walkie-talkie, replacing spaces with tabs. At first sight
you can also use empty grips to work from position-
mi inside the line, but there is a simpler solution: on each
the eighth character, we will insert the mark inside the line. Then,
when there are spaces before this mark, we will replace the corresponding
The corresponding sequence is a tab character:

```
function unexpandTabs (s, tab)
tab = tab or 8
s = expandTabs (s)
local pat = string.rep (“.”, tab)
s = string.gsub (s, pat, “% 0 \ 1”)
s = string.gsub (s, “+ \ 1”, “\ t”)
s = string.gsub (s, “\ 1”, “”)
return s
end
```

This function starts by replacing all existing
tab characters with spaces. She then builds an auxiliary
template and uses it to add a markup (manager
character \ 1) after every tab characters. Further, all successive
number of spaces followed by a mark are replaced with
tabs. Finally, all overlays are removed.

21.6. Tricky tricks

Templates are a very powerful tool for working with strings.
You can perform many complex operations with just a few
calls to string.gsub . However, like any other force, it must be
use carefully.
Using templates does not replace the parser. For quick solutions
ny (quick-and-dirty) you can use templates to work
with the source code, but the resulting solutions most likely won't
be of high quality. As an example, let's consider

Rome template that we used for search comment in
C program: '/%*.-%*/' . If you have a line in your program, so-
holding "/ *", then you may get the wrong result:

```
test = [[char s [] = "a / * here"; / * a tricky string * /]]
print (string.gsub (test, "/%*.-%*/", "<COMMENT>"))
-> char s [] = "a <COMMENT>
```

Lines with such content are quite rare, and for
your personal goals, a pattern like this will likely work.
But you cannot redistribute the program with this error.
Typically, templates work quite efficiently in Lua: my
an old Pentium computer only needs 0.3 seconds to
would find all the words in a 4.4 MB text (850K words). But always
it is better to take some precautions. It's always better to de-
make the template as accurate as possible; imprecise patterns are slower
accurate. A simple example is using '(.-)% $' for
getting the entire substring up to the first occurrence of the dollar sign. If a
there is a dollar sign in the line, then everything is fine; but let's assume that
there is no dollar sign in the string at all. Then the algorithm first
la will try to get a substring that matches the pattern to
starting from the first position within the string. Then he will move
all along the line looking for a dollar sign. When the line ends,
then we will get pattern mismatches *only for the first position*
inside the string. Then the algorithm will do the same, starting already
from the second position inside the string, etc. Thus, we get
the quadratic time complexity, taking more than 4 minutes to
my Pentium for a 100K character string. You can easily fix
fix this situation by tying the pattern to the beginning of the line with
'^ (.-)% $' . When using such a binding, the execution takes
it takes only one hundredth of a second.
Also, be very careful with *empty templates* , i.e. template
bosoms that are satisfied by the empty string. For example, if you
if you try to search for names using the '% a *' pattern , then you are everywhere
you will find names:

```
i, j = string.find ("; $% ** # $ hello13", "% a *")
print (i, j) -> 1 0
```

In this example, calling string.find correctly finds an empty string.
a sequence of letters at the beginning of a line.
You should never write a pattern that begins or begins
ends with '-' , since it will be satisfied with an empty string
ka. This modifier usually needs something around it, for

in order to limit it. Likewise templates that include
'. *' are also quite tricky, as this construct can
much more than you planned.
Sometimes it's easier to use Lua itself to build templates.
We have already used this technique in the function that transforms the
are white in tabs. As another example, let's distribute
see how we can find strings of more than 70 characters. Such
a string is a sequence of 70 or more characters
fishing other than '\ n' . Single character other than '\ n'
belongs to class '[^ \ n]' . Accordingly, we can get
pattern for a long string by repeating pattern for character 70 times and
by adding a pattern for zero or more of the following sym-
oxen. Instead of explicitly writing out this pattern, we can co-
create it with string.rep :

```
pattern = string.rep (“[^ \ n]”, 70) .. “[^ \ n] *”
```

As another example, let's say you want to do a search, an odd
case-sensitive. To do this, you can replace each letter
woo *x* in the template for the class '[*xX*]' , that is, a class that includes and
lowercase and uppercase versions of the letter. We can automate
this transformation using the following function:

```
function nocase (s)
s = string.gsub (s, “% a”, function (c)
return “[“ .. string.lower (c) .. string.upper (c) .. “]”
end)
return s
end
print (nocase (“Hi there!”)) -> [hH] [iI] [tT] [hH] [eE] [rR] [eE]!
```

Sometimes you just need to replace every occurrence of s1 with s2 ,
without considering any magic symbols. If both lines are explicitly given
in the text, then you can easily add all the necessary transformations yourself.
development for magic symbols, but if these are variables, then you
you will need additional gsubs to do this job:

```
s1 = string.gsub (s1, “(% W)”, “%%% 1”)
s2 = string.gsub (s2, “%%”, “%%%%”)
```

In the line we are looking for, we replace all non-alphanumeric-
characters, in the replacement string we replace only the '%' character .
Another useful technique for working with templates is to
completing special processing of the line before the main
work. Suppose we want to convert to uppercase all letters containing
inside double quotes, but inside the string itself
can be '\ ”' :

follows a typical string: “This is \” great \ ”!”.
One approach for such cases is coding
input string. For example, let's replace “\” ” with “ \ 1 ” . but
if the source text already contained the "\ 1" character , then we have a
lema. An easy way to do the coding and avoid this
the problem is replacing all sequences "\ *x* " with "\ *ddd* " ,
where *ddd* is the decimal representation of the character *x* :

```
function code (s)
return (string.gsub (s, “\\ (.)”, function (x)
return string.format (“\\% 03d”, string.byte (x))
end))
end
```

Now any sequence "\ *ddd* " could only come from
our encoding, since any “\ *ddd* ” in the original line is so
would be encoded. Therefore decoding is simple
task:

```
function decode (s)
return (string.gsub (s, “\\ (% d% d% d)”, function (d)
return “\\” .. string.char (tonumber (d))
end))
end
```

We can now complete our task. Since the encoded
the string no longer contains "\" " , then we can safely use
pattern "'.-'" :

```
s = [[follows a typical string: “This is \” great \ ”!”.]]
s = code (s)
s = string.gsub (s, '”.-”', string.upper)
s = decode (s)
print (s) -> follows a typical string: “THIS IS \” GREAT \ ”!”.
```

Or, writing it shorter:

```
print (decode (string.gsub (code (s), '”.-”', string.upper)))
```

21.7. Unicode

At the moment, the library for working with strings does not contain
explicit support for unicode. However, it is not difficult to implement some
some simple tasks for working with Unicode strings encoded in
UTF-8 without using additional libraries.
The primary encoding for Unicode on the Web is UTF-8. Because of her
compatibility with ASCII this encoding is also very well suited
for Lua. This compatibility ensures that a number of operating techniques
those with strings without any modification will work with UTF-8.

UTF-8 represents each unicode character with a different number
byte. For example, the character 'A' represents one byte, 65; Sim-
ox Aleph, which has Unicode code 1488, is represented by a two-byte
sequence 215-144. UTF-8 represents all characters from
ASCII as ASCII, that is, one byte with a value less than 128.
All other characters are represented by byte sequences,
where the first byte lies in the range [194, 244] and the following bytes are
you are in the range [128, 191]. More precisely, the range of the first byte for
two-byte sequences are [194, 223], for three-byte
sequences [224, 239] and for four-byte sequences
telities [240, 244]. This arrangement ensures that the
a character for any character will never be found inside the after-
sequence for another symbol. For example, a byte less than
128 will never occur in a multibyte sequence; it
always represented by its ASCII character.
In Lua, you can read, write and store strings in UTF-8
like regular strings. String constants (literals) can also
contain UTF-8 inside. (Of course, you most likely want
edit your file as a UTF-8 file.) Concatenation operation
is executed correctly for all strings in UTF-8. Comparison operations
strings (less than, less than or equal, etc.) compare strings in
UTF-8, following unicode character order.
Operating system function library and library for
I / O are really just interfaces to opera-
system, so their UTF-8 support depends on
UTF-8 on the system itself. On Linux, for example, we can use
UTF-8 for filenames, but Windows uses UTF-16. therefore
to work with file names in Unicode on Windows, you will need to
additional libraries or modification of standard libraries
tech Lua.
Let's see how functions from the library to work with
strings work with strings in UTF-8.
Functions string.reverse , string.byte , string.char , string.
upper and string.lower do not work with UTF-8 strings because
each of these functions considers one character to be one byte.
The string.format and string.rep functions work without any problems.
melt with strings in UTF-8, except for the '% c' option , which is
assumes that one character is one byte. String.len functions
and string.sub work correctly with strings in UTF-8, but at the same time

indexes no longer refer to characters, but to bytes. Often
this is exactly what you need. But we can easily count the number
characters, as we'll see shortly.
For functions for working with templates, their applicability depends on
template. Simple templates work without any problems, since
representation of one character can never occur inside
When representing another character. Character classes and sets
characters only work for ASCII characters. For example, the template
"% S" works for UTF-8 strings, but it will only match
ASCII spaces and will not match unicode spaces, so
kim as unbreakable space (U + 00A0), paragraph separator
(U + 2029) or Mongolian G + 180E.
Some templates can make good use of features
UTF-8. For example, if you want to count the number of characters in a line
ke, then you can use the following expression:

```
# (string.gsub (s, "[\ 128- \ 191]", ""))
```

In this example, gsub strips out the second, third and fourth bytes,
leaving one byte for each character as a result.
Similarly, the following example shows how to iterate over
all characters in a string are in UTF-8:

```
for c in string.gmatch (s, ". [\ 128- \ 191] *") do
print (c)
end
```

Listing 21.1 shows some tricks for working with UTF-8
strings in Lua. Of course, to run these examples you need
platform where print supports UTF-8.
Unfortunately, Lua has nothing more to offer. Adequate
Native Unicode support requires huge tables, which is bad
correspond to the small size of Lua. Unicode has many features.
tei. It is almost impossible to abstract any concept
from specific languages. Even the concept of what a symbol is is very
fuzzy, since there is no one-to-one correspondence between
Unicode-encoded characters and graphemes (that is,
lamas with diacritics and "completely ignored"
symbols). Other seemingly basic concepts such as what is
symbol are also different for different languages.
What, in my opinion, is missing in Lua is the functions for translation
between UTF-8 and unicode and validating strings in UTF-8.
Perhaps they will be included in the next version of Lua. For the rest of the
the best option would be to use an external library

like Slnunicode.

Listing 21.1. Examples of working with UTF-8 in Lua

```
local a = {}
a [#a + 1] = “Nähdään”
a [#a + 1] = “ação”
a [#a + 1] = “ÃØÆËl”
local l = table.concat (a, “;”)
print (l, # (string.gsub (l, “[\ 128- \ 191]”, “”)))
-> Nähdään; ação; ÃØÆËÐ 18
for w in string.gmatch (l, “[^;] +”) do
print (w)
end
-> Nähdään
-> ação
-> ÃØÆËÐ
for c in string.gmatch (a [3], “. [\ 128- \ 191] *”) do
print (c)
end
-> Ã
-> Ø
-> Æ
-> Ë
-> Ð
```

Exercises

Exercise 21.1. Write a split function that gets string and delimiter pattern and returns a sequential number of blocks separated by separator:

```
t = split (“a whole new world”, “”)
- t = {“a”, “whole”, “new”, “world”}
```

How does your function handle empty lines? (In particular, is the empty string an empty sequence, or sequence with one blank line?)

Exercise 21.2. The patterns '% D' and '[^% d]' are equivalent. What what about the patterns '[^% d% u]' and '[% D% U]' ?

Exercise 21.3. Write a function for transliteration. This the function gets a string and replaces every character in this string with another character in accordance with the table given second argument. If the table maps 'a' to 'b' then the function should replace every occurrence of 'a' with 'b' . If a the table maps 'a' to ***false*** , then the function should delete all occurrences of the 'a' character from the string.

Exercise 21.4. Write a function that reverses
a string in UTF-8.
Exercise 21.5. Write a transliteration function for UTF-8.

Chapter 22
Library input / output

The I / O library provides two different models for
working with files. The simple model uses the *current input* and
the current output files, and all its operations are performed on these
files. The full model uses explicit file pointers; it
relies on an object-oriented approach that defines
Lets all operations as methods on file pointers.
A simple model is convenient for simple things; we used
her throughout the book. But it is not enough for more flexible
work with files, for example, for simultaneous reading or one-
temporary recording in several files at once. For this we need
complete model.

22.1. Simple model input / output

A simple model performs all its operations on the two current files
lami. The library uses standard input when initializing
(stdin) as default input file and standard output (stdout)
as the default output file. Thus, when we do
something like io.read () that we read from standard input.
We can change these current files using the functions
io.input and io.output . Calling io.input (filename) opens the specified
file to read and sets it as input file
default. From now on, all input will come from this-
th file until the next call to io.input ; io.output works similarly
Generally, but for output. On error, both functions call
mistake. If you want to explicitly handle errors, then you need
complete model.
The write function is simpler than read , so we'll first look at
her. The io.write function receives an arbitrary number of string ar-
guments and writes them to the default output file. She pre-
converts numbers to strings using standard conversion rules

vania; for complete control over this conversion use
string.format function :

```
> io.write ("sin (3) =", math.sin (3), "\ n")
-> sin (3) = 0.14112000805987
> io.write (string.format ("sin (3) =% .4f \ n", math.sin (3)))
-> sin (3) = 0.1411
```

Avoid code like io.write (a..b..c) ; call io.write (a, b, c)
does the same thing with fewer resources, since
it avoids the concatenation operation.
Use print for small programs or for debugging and
write when you need full control over the output:

```
> print ("hello", "Lua"); print ("Hi")
-> hello Lua
-> Hi
> io.write ("hello", "Lua"); io.write ("Hi", "\ n")
-> helloLuaHi
```

Unlike print , the write function does not add any
characters like tabs or move to next
string. In addition, the write function allows you to redirect your
output, whereas print always uses standard output. At the same time
net, print automatically applies tostring to its arguments;
this is useful for debugging, but it can hide errors if you are not aware
are thoughtful to the conclusion.
The io.read function reads lines from the current input file. Her
arguments control what to read:

"* A"
Reads the entire file
"* L"
Reads the next line (no line feed)
"* L"
Reads the next line (with a line feed)
"* N"
Reads a number
num
Reads a string of no more than *num* characters

The io.read ("* a") call reads the entire current input file, starting
from the current position. If we are at the end of the file or the file is empty,
then the call returns an empty string.
Since Lua works efficiently with long strings,
an easy way to write filters in Lua is to read
the whole file into a line, perform line processing (usually using
shchi gsub) and then write the line to the output:

```
t = io.read ("* a")
```

```
- read the entire file
t = string.gsub (t, ...)
- do the job
io.write (t)
- write file
```

As an example, the following piece of code is a law-
chennaya program for encoding file content in MIME
quoted-printable . Each non-ASCII byte is encoded as = *xx* , where *xx* is
it is the hexadecimal byte value. For the integrity of the coding
The equality symbol itself must also be encoded:

```
t = io.read (“* a”)
t = string.gsub (t, “([\ 128- \ 255 =])”, function (c)
return string.format (“=% 02X”, string.byte (c))
end)
io.write (t)
```

The pattern used in gsub finds all bytes from 128 to 255,
including the equal sign.
The io.read (“* l”) call reads the next line from the current input-
leg file without line feed character ('\ n'); io.read call (“* L”)
is similar, but only it returns a newline character (if
he was present). When we reach the end of the file, the function
rotates ***nil*** (since there are no more lines). Pattern “* l” for function
read is the default. I usually use this
pattern only when it naturally processes the file line by line
coy; otherwise I prefer to read the entire file at once when
help "* a" or read it in blocks, as we'll see later.
As a simple example of using this pattern, follow-
This program copies the current input to the current output by numbering
with each line:

```
for count = 1, math.huge do
local line = io.read ()
if line == nil then break end
io.write (string.format (“% 6d“, count), line, “\ n”)
end
```

However, in order to iterate over the entire file, line by line,
it is better to use the io.lines iterator . For example, we can write
Put the complete program for sorting the lines of the file as follows -
in a way:

```
local lines = {}
- read lines into table 'lines'
for line in io.lines () do lines [#lines + 1] = line end
- we sort
```

```
table.sort (lines)
- write all lines
for _, l in ipairs (lines) do io.write (l, "\ n") end
```

The io.read ("* n") call reads a number from the current input file. This is the only case where the read function returns a number, not a string. When a program needs to read a lot of numbers from a file, then the absence of intermediate lines improves performance. Op- ration * n skips all spaces before the number and supports such number formats like -3 , +5.2 , 1000 and -3.4e-23 . If the function is not can find a number at the current position (due to incorrect format or end of file), it returns ***nil*** .

You can call read by passing multiple options at once; for every For the second argument, the function will return the corresponding value. Let u you have a file containing three numbers per line:

```
6.0 -3.23 15e12
4.3 234 1000001
...
```

Now you need to print the maximum for each line. You can You can read all three numbers in one read call :

```
while true do
local n1, n2, n3 = io.read ("* n", "* n", "* n")
if not n1 then break end
print (math.max (n1, n2, n3))
end
```

Besides the standard templates, you can call read by passing in number *n* as argument : in this case read tries to read *n* characters from the input file. If she can't read one character (end of file), it returns ***nil*** ; otherwise a string with at most *n* characters is returned . As an example The following program demonstrates an efficient way (for Lua, of course) copy the file from stdin to stdout :

```
while true do
local block = io.read (2 ^ 13) -- 8K buffer size
if not block then break end
io.write (block)
end
```

As a separate case, read (0) works as an end-of-file check: it returns an empty string if there are characters in the file, and ***nil*** if end of file reached.

22.2. Full model input / output

For more control over I / O you can use
complete model. The key concept in this model is *indicating*
Tel file (file handle), which is similar to the FILE * to C: he presented
Lets an open file at the current location.
To open a file, use the io.open function ,
which is analogous to the fopen function in C. As arguments, it
takes a filename and a string specifying the mode. This line can
contain 'r' for reading, 'w' for writing (writing erases the previous
the contents of the file) or 'a' to append to the file, also it
may contain 'b' to work with binaries. Function
open returns a new file pointer. On error open
returns ***nil*** as well as the error message and error code:

```
print (io.open (“non-existent-file”, “r”))
-> nil non-existent-file: No such file or directory 2
print (io.open (“/ etc / passwd”, “w”))
-> nil / etc / passwd: Permission denied 13
```

The interpretation of error codes is system dependent.
A typical error checking method is as follows:

```
local f = assert (io.open (filename, mode))
```

If an error occurs, the error message appears as a second
The last argument to assert , which prints this message.
After you open the file, you can read and write from it
into it using the read / write methods . They are similar to functions
read / write , but you call them as file pointer methods, using
using the colon. For example, in order to open a file and
honor all of it, you can use the following code:

```
local f = assert (io.open (filename, “r”))
local t = f: read (“* a”)
f: close ()
```

The I / O library provides three predefined
pointers to standard files in C: io.stdin , io.stdout, and

io.stderr . Therefore, you can send the error message directly
to the appropriate standard file:

```
io.stderr: write (message)
```

You can use the complete model along with the simple model.
To get a pointer to the current input file,
blows call io.input () with no arguments. In order to ask the
file locator as the current input file, call

io.input (hanle) (similar calls work for io.output as well).
For example, if you want to temporarily change the current input
file, then you can write something like the following:

```
local temp = io.input ()
- save the current file
io.input ("newinput")
- open new current file
<process input>
io.input (): close ()
- close the current file
io.input (temp)
- restore the previous file
```

Instead of io.read to read from a file, we can also use
Vat io.lines . As we saw in the previous examples, io.lines
returns an iterator that reads sequentially from a file.
The first argument to io.lines can be a file name or a specified
tel per file. If a filename was passed, then io.lines will open
the file is in read-only mode and will close the file after reaching the end
file. If a file pointer was passed, then io.lines will be used.
use this file for reading; in this case io.lines won't
close the file when it reaches the end. In the case of a call at all
with no arguments, io.lines will read data from the current input
file.
Since Lua 5.2, the io.lines function also accepts the same
same options as io.read (after the first argument). As an adjunct
measure the following code copies the file using io.lines :

```
for block in io.lines (filename, 2 ^ 13) do
io.write (block)
end
```

A small trick to increase speed

It is usually faster in Lua to read an entire file than to read a line of it.
by line. However, sometimes we come across a large file (at-
example, tens or even hundreds of megabytes), read which in its entirety
it would be inappropriate. If you want to get the maximum
performance when working with such large files, it is faster
will read it in large enough blocks (for example, by
8K). To avoid a possible line break, you can

just ask to read one more line:
local lines, rest = f: read (BUFSIZE, “* l”)
The rest variable will get the remainder of any line broken when
reading a block. Then we combine the block and the resulting remainder. Ta-
This way the block will always end at line boundaries.
The example in Listing 22.1 uses this technique to implement wc ,
a program that counts the number of characters, words, and lines in a file.
Note the use of io.lines to implement
iterations and the "* L" option to read a line, this is available starting at
Lua 5.2.
Listing 22.1. Wc program
```
local BUFSIZE = 2 ^ 13 - 8K
local f = io.input (arg [1])
- open input file
local cc, lc, wc = 0, 0, 0
- counters
for lines, rest in io.lines (arg [1], BUFSIZE, “* L”) do
if rest then lines = lines .. rest end
cc = cc + #lines
- count the words in the block
local _, t = string.gsub (lines, “% S +”, “”)
wc = wc + t
- count '\ n'
_, t = string.gsub (lines, “\ n”, “\ n”)
lc = lc + t
end
print (lc, wc, cc)
```

24.2. Hooks

The trap mechanism allows us to register a function that
paradise will be called upon the occurrence of certain events during
program execution. There are four types of events that
can trigger traps:
• a *call* event occurs when Lua calls a function;
• the *return* event occurs when the function returns;
• a *line* event occurs when Lua starts execution
next line;
• the *counter* event occurs after a specified number of
mand.
Lua calls hooks with a single argument, a string,
the event that led to the call: “call” (or “tail

call ”), “ return ” , “ line ” or “ count ” . For line event also
the second argument is passed, the new line number. To get up to-
additional information inside the trap should be used
debug.getinfo .
To register a trap, we call the function
debug.sethook with two or three arguments: the first argument is
this is the corresponding function; the second argument is a mask string,
describing exactly what events we want to track, and not
the required third argument is a number that specifies how often
that we want to receive counter events. In order to track
to add call, return and string events, we add the letters 'c' , 'r'
and 'l' to the mask string. To track counter events, we simply
we pass the counter as the third argument. To remove all traps
just call sethook with no parameters.
As an example, the following code sets up a primitive
a trap, which for each next run of the print run-
her number:

```
debug.sethook (print, “l”)
```

This call sets print as a hook function and sets it
call only for row events. More complex hook function
can use getinfo to add a name to the output
current file:

```
function trace (event, line)
local s = debug.getinfo (2) .short_src
print (s .. “:” .. line)
end
debug.sethook (trace, “l”)
```

A useful function to use in traps is
debug.debug . This simple function prints a prompt, reads from
input and then executes the given commands. It is roughly equivalent to
tape on the following code:

```
function debug1 ()
while true do
io.write (“debug>“)
local line = io.read ()
if line == “cont” then break end
assert (load (line)) ()
end
end
```

When the user enters “cont” at the prompt , this
the function ends. The standard implementation is very simple and you

executes commands in the global environment outside of the code being debugged.
Exercise 24.5 discusses a better implementation.

24.3. Profiling

Despite its name, the debug library is also useful for
not only debugging tasks. A typical such task is
profiling (obtaining information about the time spent
on the execution of this or that piece of code). For profiling
time, it is better to use the C-interface: the cost of the call
each Lua hook is quite high and can greatly distort
results. However, for simple profiling that counts
Many times, the Lua code is fine. In this section, we write
we have the simplest profiler, which for each called
the function will tell you how many times it was called during the execution
the program.
The main data structure in our program will be two tables
ts: one maps functions to their counters, and the second maps
functions their names. The indexes for both of these tables will be
The functions themselves can act.

```
local Counters = {}
local Names = {}
```

We can extract the function names after profiling, but we
we get better results if we get the names of the functions,
while they are active, since in this case Lua can look at
looking up the name of the function, the calling code.
Now let's define a hook function. Its task is
get the called function and increment the corresponding count-
chick, it also collects function names:

```
local function hook ()
local f = debug.getinfo (2, “f”). func
local count = Counters [f]
if count == nil then -– is the function 'f' called the first time?
Counters [f] = 1
Names [f] = debug.getinfo (2, “Sn”)
else
- only increase the counter value
Counters [f] = count + 1
end
end
```

The next step is to run the program with this hook.
We will assume that the main block of the program is in the file
and the name of this file is passed as an argument to the profiling program-
to the box:

```
% lua profiler main-prog
```

Then the profiler can take the filename from arg [1] , installed
hook a trap and execute the file:

```
local f = assert (loadfile (arg [1]))
debug.sethook (hook, "c") -- set a hook
f ()
- execute the profiled program
debug.sethook ()
- disable the trap
```

The final step is to actually display the results. Funk-
tion getname Listing 24.2 outputs for each function sootvetst-
name. To avoid confusion, to each name
add the place of the corresponding function in the form *file: string* .
If the function has no name, then we only print the location. If the function

is a function in C, then we use only its name (so
as she has no place). With this in mind, below is the code that I print-
containing information about calls:

```
for func, count in pairs (Counters) do
print (getname (func), count)
end
```

Listing 24.2. Getting the function name

```
function getname (func)
local n = Names [func]
if n.what == "C" then
return n.name
end
local lc = string.format ("[% s]:% d", n.short_src, n.linedefined)
if n.what ~ = "main" and n.namewhat ~ = "" then
return string.format ("% s (% s)", lc, n.name)
else
return lc
end
end
```

If we apply our profiler to the example with the chain Mar-
cova from section 10.3, then we get something like:

```
[markov.lua]: 4 884723
write 10000
[markov.lua]: 0 1
```

```
read 31103
sub 884722
[markov.lua]: 1 (allwords) 1
[markov.lua]: 20 (prefix) 894723
find 915824
[markov.lua]: 26 (insert) 884723
random 10000
sethook 1
insert 884723
```

This shows that the anonymous function on line 4 (which is is our iterator defined inside `allwords`) was called 884,723 times, the `write` (`io.write`) function was called 10,000 times, etc.

This profiler can be improved, for example, add a dirty output customization, improved function name printing, etc. less even this profiler is already useful and can be used Called as a basis for writing more advanced tools.

Exercises

Exercise 24.1. Why recursion in `getvarvalue` function (Listing 24.1) will it stop?

Exercise 24.2. Modify the `getvarvalue` function (lis-Thing 24.1) to work with various coroutines (like other functions from the debug library).

Exercise 24.3. Write a `setvarvalue` function .

Exercise 24.4. Based on the `getvarvalue` function write `getallvars` function that returns a table with all variables that are visible at a given location (return-May the table should not include environment variables it should instead inherit them from the original arms).

Exercise 24.5. Write an improved version of `debug.debug` , which executes the given commands as if they were executed in the scope of the calling function. (*Hint* : run commands in empty environment and use the `__index` function as a metamethod `getvarvalue` .)

Exercise 24.6. Modify the previous example to you could change variables.

Exercise 24.7. Implement some of the suggested improvements
for the profiler from Section 24.3.
Exercise 24.8. Write a library to work with points
breakpoint. She should offer at least
two functions:
setbreakpoint (function, line) -> returns handle
removebreakpoint (handle)
The breakpoint is set by a function and a line within the function.
When execution reaches a breakpoint, you should exit
name debug.debug .
(*Hint* : for the simplest implementation, use the trap
strings and a hook function that checks if we hit
to a breakpoint; to improve performance, we can
turn on this trap only when we are inside
the function we are teasing.)

Part IV
With API

CHAPTER 25
C API overview

Lua is an *embedded language* . This means Lua is not separate.
package, and a library that we can link to other
instructions to add Lua features to them.
You may be wondering if Lua is not a standalone program, but
so far in the book, we have used Lua as a standalone program.
Rammu. The solution to this question is the Lua interpreter (executing
nimy file the lua). This interpreter is a small application
ny (less than five hundred lines of code) that uses
the Lua library in order to implement a separate interpretation
Lua torus. This program deals with user interaction
lem, taking files and lines and passing them to the Lua library, which

does basic work (such as running Lua code).
This ability to use the library in order to expand
rit application possibilities - this is exactly what makes Lua *distribution widening tongue* . At the same time, the program that is used
calls Lua, can register new functions in the Lua environment,
thus adding features that could not be written
us on Lua itself. This is what makes Lua an *extensible language* .
These two views of Lua (as an extension language and as a
extensible language) correspond to two types of interaction between
C and Lua. In the first case, C controls, and Lua is just a library.
We call the C code corresponding to this type of interaction
it *application code* (application code). In the second case, the control
in Lua, and C is a library. In this case, the C code is called *bib-library code* . Both of these types of code use the same API
for interacting with Lua, the so-called C API.
The C API is simply a collection of functions that enable C code
interact with Lua [1] . It includes functions for reading

and writing global Lua variables to call functions in Lua,
to execute snippets of Lua code, to register functions
in C, so that they can then be called from Lua code, etc. Practice
Technically anything that Lua code can do can be done in C.
using the C API.
The C API follows the C style, which differs markedly from the C API.
for the Lua language. When we program in C, we follow the types
data, error handling, memory allocation errors and
other difficult places. Most API functions do not validate
the validity of their arguments; it is your job to make sure that
arguments are correct before function call [2] . If you admit
error, you will most likely get an error like "segmentation fault"
or something like that instead of a nice error message. More
Moreover, the C API emphasizes flexibility and simplicity, often at the expense of easy
use bones. Typical tasks may require several
these API calls. It can be tedious, but it gives you
complete control over what is happening.
As the title suggests, the purpose of this chapter is to
necessary when using Lua from C. Don't try to understand now
all the details of what is happening. We will dwell on this later. but

do not forget that you can always find additional information
See the Lua Reference Manual. Moreover, you can find
some examples of using the API in the Lua distribution itself. Department
the lua Lua interpreter (lua.c) gives examples of application code,
while the standard libraries (lmathlib.c , lstrlib.c
etc.) provide examples of library code.
From now on, we act as C programmers.
When I talk about "you", I mean exactly you, I program
on S.
An important component in the communication between Lua and C is
a permanently present virtual *stack* . Almost all functions
APIs work with values on this stack. All data exchange between
do Lua and C go through this stack. What's more, you can also
use this stack to store intermediate results.
This stack allows you to solve problems with a fundamental difference -
mi between Lua and C: the first difference is that Lua has
garbage collection, while C is explicit memory management; second
the difference is the difference between dynamic typing in
Lua and static typing in C. We will discuss the stack in more detail.
in section 25.2.

25.1. First example

We'll start this overview with an example of a simple application: a stand-alone
Lua interpreter. We can write a primitive interpreter
The Lua tool, as shown in Listing 25.1. Lua.h header file
defines the basic functions provided by Lua. It includes in
functions to create a new Lua environment, to call a function
tions in Lua (such as lua_pcall) to read and write global
variables in the Lua environment, to register new functions, which
which can be called from Lua, etc. Anything defined in the file
lua.h is prefixed with _lua .

Listing 25.1. Simple standalone Lua interpreter

```
#include <stdio.h>
#include <string.h>
#include “lua.h”
#include "lauxlib.h"
#include "lualib.h"
int main (void) {
char buff [256];
```

```
int error;
lua_State * L = luaL_newstate (); / * opens Lua * /
luaL_openlibs (L); / * opens standard libraries * /
while (fgets (buff, sizeof (buff), stdin)! = NULL) {
error = luaL_loadstring (L, buff) || lua_pcall (L, 0, 0, 0);
if (error) {
fprintf (stderr, “% s \ n”, lua_tostring (L, -1));
lua_pop (L, 1); / * pop the error message off the stack * /
}
}
lua_close (L);
return 0;
}
```

The lauxlib.h header file defines the functions provided by nye *additional libraries* (auxiliary library). All definitions from that file start with luaL_ (for example, luaL_loadstring). The additional library uses the basic API provided by ny lua.h to provide a higher level of abstraction, in particular the abstractions used by the standard libraries. The core API strives for economy and orthogonality, while while the additional library strives for practicality for common tasks. Of course this is easy for your program too can create the necessary abstractions. Keep in mind that before- the additional library does not have access to Lua internals. Everything, that it does, it does through the standard API. What does she do, maybe do your program too.

The Lua library does not define any global re- men. It stores all its state in a dynamic structure. lua_State ; all functions inside Lua receive a pointer to this structure round as an argument. This implementation makes Lua reentrant noisy and ready for use in multi-strand applications.

As its name suggests , the luaL_newstate function creates a new Lua state. When luaL_newstate creates a new state, it does not contains no built-in functions, not even print . In order to keep Lua small, all standard libraries are represented as separate packages, so you are not required to use them if you don't need them. The lualib.h header file defines the function tions for opening libraries. LuaL_openlibs function opens all standard libraries.

After creating a state and populating it with standard libraries By the way, it's time to start doing user input. For each

the line that the user enters, the program first calls
luaL_loadstring to compile the injected code. If mistakes
no, then this call returns zero and places the resulting function
per stack. (Remember that we will discuss the stack in detail in the following -
See the next section.) The program then calls lua_pcall , which
pops a function off the stack and executes it in protected mode.
Like luaL_loadstring , lua_pcall returns zero if
no mistakes. In case of an error, both functions post a message about
an error on the stack; we will receive this message using the function
lua_tostring , and after we print it, we remove it from
stack using the lua_pop function .
Please note that in case of an error, the program simply prints
melts the error message to standard stream for errors. Infusion
how error handling in C can be quite complex, and how
should be followed, often depends on the type of your application. Nucleus
Lua itself does not print anything to any stream (file); it in case
error simply returns an error message. Each application
ny can process these messages in the most appropriate way for
him way. For simplicity, we will use the following ob-
an error worker who, in case of an error, prints a message about
error, closes the Lua state and exits the application:

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
void error (lua_State * L, const char * fmt, ...) {
va_list argp;
va_start (argp, fmt);
vfprintf (stderr, fmt, argp);
va_end (argp);
lua_close (L);
exit (EXIT_FAILURE);
}
```

We'll come back to handling errors in the application code later.
Since you can compile Lua both as C code and as code
in C ++, lua.h does not include the following standard amendment,
common in C code:

```
#ifdef __cplusplus
extern “C” {
#endif
...
#ifdef __cplusplus
}
#endif
```

If you are compiling Lua as C code (the most common
case) and use it in C ++, you can include lua.hpp
instead lua.h . It is defined as follows:

```
extern “C” {
#include “lua.h”
}
```

25.2. Stack

When passing values between Lua and C, we are faced with two complexities
ness: mismatch between static and dynamic systems
mami typing and mismatch between automatic and manual
memory management.
In Lua, when we write a [k] = v , variables can have very different
types, even a may have a different type (due to the use of
metatables). However, if we want to provide this operation in C,
then any settable function must be of a fixed type. Us
dozens of functions will be needed for this simple operation (one by one
functions for each combination of the types of three arguments).
We can solve this problem by introducing a new type - union os-
new types, let's call it lua_Value , which can represent
all values in Lua. Then we can declare settable as follows
way:

```
void lua_settable (lua_Value a, lua_Value k, lua_Value v);
```

However, this solution has two disadvantages. First, maybe
it can be very difficult to map a complex data type to others
languages; we designed Lua to interoperate easily
not only with C / C ++, but also with Java, Fortran, C #, etc. Secondly,
Lua does garbage collection: if we store a Lua table in
variable C, then Lua itself cannot know about this in any way and can (error
sideways) decide that this table is no longer needed and delete it.
Therefore, the Lua API does not define anything like the lua_Value type .
Instead, it uses an abstract stack to exchange values-
mi between Lua and C. Each slot in this stack can contain any
Lua value. When we want to get a value from Lua (for example,
the value of a global variable), you call Lua and it puts
pushes a value onto the stack. When you want to pass a Lua value, then you
first push the value onto the stack and then call Lua (which
pops that value off the stack). We still need different

functions to push each type of C onto the stack and each
type on C to remove from the stack, but we no longer have a combinatorial
increasing the number of functions as before. Moreover, since this
the stack lives inside Lua, then the garbage collector knows what values
uses C.
Almost all functions in the API use a stack. As we already see-
In our first example, the luaL_loadstring function leaves
its result on the stack (either as a compiled block, or
as an error message); lua_pcall takes the called function
off the stack and leaves any error message on the stack.
Lua works with the stack strictly in accordance with the LIFO principle
(Last In, First Out). When you call Lua, it only changes
the top of the stack. C code has more freedom; in particular, he can
view any item on the stack, as well as insert and remove
elements from anywhere in the stack.

Pushing items onto the stack

The API contains one function to push on the stack each
th type C, which can be represented in Lua: lua_pushnil for
constants ***nil*** , lua_pushboolean for boolean values (integers
sat in C), lua_pushnumber (for double), lua_pushinteger for integers
numbers, lua_pushunsigned for unsigned integers, lua_pushl-
string for arbitrary strings (pointer to char and length) and lua_
pushstring for regular ASCIIZ strings:

```
void lua_pushnil (lua_State * L);
void lua_pushboolean (lua_State * L, int bool);
void lua_pushnumber (lua_State * L, lua_Number n);
void lua_pushinteger (lua_State * L, lua_Integer n);
void lua_pushunsigned (lua_State * L, lua_Unsigned n);
void lua_pushlstring (lua_State * L, const char * s, size_t len);
void lua_pushstring (lua_State * L, const char * s);
```

There are also functions for pushing functions on the stack to
C and objects of type *userdata* , but we'll look at them later.
The lua_Number type is a numeric type in Lua. By default this is the type
double , but it can be changed to float or even long int for different
personal architectures. The lua_Integer type is an integer type with
sign large enough to hold the size of large
lines. It is usually defined as ptrdiff_t . The lua_Unsigned type is
32-bit unsigned C integer type; used by the library

for bitwise operations and various functions.
Lua strings are not null terminated, they can contain
Reap arbitrary binary data. Accordingly, they should
on an explicit length. The main function for the building is
ki on the stack is lua_pushlstring , requiring an explicit assignment
the length of the string. For null terminated strings you can
use the lua_pushstring function , which to calculate
string length uses strlen . Lua never stores pointers
to external lines (or to any other external object, except
functions in C). For any line that needs to be stored
thread, Lua either makes a copy or reuses an existing one.
Accordingly, you can free or modify your buffer like
only control will return from these functions.
When you push an item onto the stack, it is your responsibility -
make sure that there is enough space on the stack for it. Pom-
that you are now a C programmer. When Lua starts
to be executed and whenever Lua calls C, there is a
at least 20 free slots. (The lua.h header file is op-
modifies this constant as LUA_MINSTACK .) Usually more than
enough, so you usually don't have to think about it. but
some tasks require more stack space, in particular
if you push items onto the stack in a loop. In these cases, you can
you can call the lua_checkstack function , which checks if
required free space on the stack:

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

Referring to elements

The API uses *indexes* to refer to items on the stack . Per-
the first element pushed onto the stack has index 1, the next one is in
dex 2, etc. We can also refer to the elements of the stack using
assuming the top of the stack as a starting point, in this case we use
negative indices. In this case, -1 corresponds to an element on
the top of the stack (that is, the last element pushed onto the stack),
–2 matches the previous item, and so on. For example, calling lua_
tostring (L, -1) returns the value at the top of the stack as a string.
As we will see below, there are cases when it is natural to refer to
stack, starting from the bottom of the stack (that is, using positive

dex), and there are cases when it is natural to use negative indices.

To check if an element is the value of a given type, the API offers a set of lua_is * functions , where * can be any a common type in Lua. Accordingly, there are functions lua_isnumber , lua_ isstring , lua_istable , etc. All of these functions have the same common prototype:

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

In fact lua_isnumber does not check whether the knowledge reading a number, but checks if the value can be converted to number; lua_isstring behaves similarly: in particular, any clean lo satisfies lua_isstring .

There is also a lua_type function that returns the type of an element. cop on the stack. Each type is represented by a constant defined in the lua.h header file : LUA_TNIL , LUA_TBOOLEAN , LUA_TNUMBER, LUA_TSTRING , LUA_TTABLE , LUA_TTHREAD , LUA_TUSERDATA and LUA_ TFUNCTION . We usually use this function in conjunction with the opera- Rathore switch statement . It is also useful when we need check if a value is a number or a non-cast string types.

To get values from the stack, there are lua_to * functions :

```
int lua_toboolean
(lua_State * L, int index);
const char * lua_tolstring (lua_State * L, int index,
size_t * len);
lua_Number lua_tonumber (lua_State * L, int index);
lua_Integer lua_tointeger (lua_State * L, int index);
lua_Unsigned lua_tounsigned (lua_State * L, int idx);
```

Lua_toboolean function converts any value to boolean some value in C (0 or 1), while using the Lua rules for boolean constructs: ***nil*** and ***false*** give 0, all others - the value 1. Any of the lua_to * functions can be called , even when the value is of the wrong type. Lua_toboolean function works for values any type; lua_tolstring returns NULL for non-string values cheniy. Numeric functions have no way of reporting errors ke, so they simply return zero on error. Usually lua_isnumber should be called to check the type, but in Lua 5.2 we introduced the following functions:

```
lua_Number lua_tonumberx (lua_State * L, int idx, int * isnum);
lua_Integer lua_tointegerx (lua_State * L, int idx, int * isnum);
lua_Unsigned lua_tounsignedx (lua_State * L, int idx, int * isnum);
```

The isnum parameter returns a boolean value indicating
about whether the corresponding Lua value was a number. (If you need this
no value is needed, then you can as the last parameter
pass NULL . The old lua_to * functions are now implemented as
macros based on these functions.)
Lua_tolstring function returns pointer to internal
a copy of the string and remembers the length of the string via len parameter .
You do not
you can change this internal copy (the const specifier reminds
tells you about it). Lua ensures that this pointer is valid until
as long as the corresponding line is on the stack. When the function
in C, a call from Lua returns, then Lua clears the stack;
so never cast pointers to Lua strings outside of a function,
received them.
Any string that lua_tolstring returns always has
a null byte at the end, but it can also contain null bytes
inside yourself. The actual size of the line is returned through the third
argument len . In particular, assuming that the value at the top of the stack is
this is a string, the following asserts are always true:

```
size_t l;
const char * s = lua_tolstring (L, -1, & l); / * any Lua string * /
assert (s [l] == '\ 0');
assert (strlen (s) <= l);
```

You can call lua_tolstring with a third parameter equal to
NULL if you don't need the length of the string. Or you can use
use the macro lua_tostring , which is actually lua_
tolstring with third parameter equal to NULL .
In order to illustrate the use of these functions
ttion Listing 25.2 shows a useful helper function,
which prints the contents of the stack. This function traverses the entire stack
bottom to top, printing each item according to its type.
Strings are printed in quotes, numbers are formatted as '% g' ,
for other values (functions, tables, etc.), only
a type. (The lua_typename function translates a numeric value that is
specifying the type, into a string.)

Listing 25.2. Printing the contents of a stack

```
static void stackDump (lua_State * L) {
int i;
int top = lua_gettop (L); / * stack depth * /
for (i = 1; i <= top; i ++) {/ * repeat for each level * /
```

```
int t = lua_type (L, i);
switch (t) {
case LUA_TSTRING: {/ * strings * /
printf (“'% s'”, lua_tostring (L, i));
break;
}
case LUA_TBOOLEAN: {/ * booleans * /
printf (lua_toboolean (L, i)? “true”: “false”);
break;
}
case LUA_TNUMBER: {/ * numbers * /
printf (“% g”, lua_tonumber (L, i));
break;
}
default: {/ * other values * /
printf (“% s”, lua_typename (L, t));
break;
}
}
printf (““); / * put a separator * /
}
printf (“\ n”); / * end the listing * /
}
```

Other stack operations

In addition to the previous functions used to exchange data between C and Lua, the API also provides the following functions to work with stack:

```
int lua_gettop (lua_State * L);
void lua_settop (lua_State * L, int index);
void lua_pushvalue (lua_State * L, int index);
void lua_remove (lua_State * L, int index);
void lua_insert (lua_State * L, int index);
void lua_replace (lua_State * L, int index);
void lua_copy (lua_State * L, int fromidx, int toidx);
```

Lua_gettop function returns the number of items on the stack, so is equal to the index of the element at the top of the stack. Lua_settop function sets the number of items on the stack. If the previous value is the top of the stack was larger, then the extra values are thrown away Xia. Otherwise, use as missing values is ***nil*** . In particular, lua_settop (L, 0) clears the entire stack. In function lua_settop you can also use negative indices.

In particular, the API provides the following macro, which removes
there are n elements from the stack :

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

The lua_pushvalue function pushes a copy of the item with the given
ny index; the lua_remove function removes an element with a given
dex, while shifting all other elements; lua_insert pereme-
pops an element from the top of the stack to the given position, while shifting
elements to free up space; lua_replace removes the value from
top of the stack and sets it to the value of the element with the given
index; finally, lua_copy copies the value at one index to
the value at a different index without changing the original value. Reverse
Note that the following operations do not affect a non-empty stack:

```
lua_settop (L, -1); / * set vertex to current value * /
lua_insert (L, -1); / * move element from top to top * /
lua_copy (L, x, x); / * copy the element to its position * /
```

The program in Listing 25.3 uses the stackDump function (defined
shown in Listing 25.2) to illustrate these stack operations.

Listing 25.3. An example of working with a stack

```
#include <stdio.h>
#include “lua.h”
#include "lauxlib.h"
static void stackDump (lua_State * L) {
<same as in Listing 25.2>
}
int main (void) {
lua_State * L = luaL_newstate ();
lua_pushboolean (L, 1);
lua_pushnumber (L, 10);
lua_pushnil (L);
lua_pushstring (L, “hello”);
stackDump (L);
/ * true 10 nil 'hello' * /
lua_pushvalue (L, -4); stackDump (L);
/ * true 10 nil 'hello' true * /
lua_replace (L, 3); stackDump (L);
/ * true 10 true 'hello' * /
lua_settop (L, 6); stackDump (L);
/ * true 10 true 'hello' nil nil * /
lua_remove (L, -3); stackDump (L);
```

```
/ * true 10 true nil nil * /
lua_settop (L, -5); stackDump (L);
/ * true * /
lua_close (L);
return 0;
}
```

25.3. C API error handling

All structures in Lua are dynamic: they grow as
necessary and reduced in size when possible. This means
thinks that in Lua we are constantly faced with a possible error when
memory allocation. Almost any operation over time can
lead to this. Moreover, many operations can cause other
many mistakes; for example, accessing a global variable can
lead to a call to the __index metamethod , and this metamethod can call
mistake. Finally, operations that allocate memory over time
call the garbage collector, which calls finalizers that
They can also lead to errors. In short, overwhelming
most functions in Lua can lead to errors.
Instead of using error codes in its API, Lua uses
exceptions for reporting errors. Unlike C ++ or Java,
the C language does not contain a mechanism for working with exceptions. For
to get around this problem, Lua uses the setjmp function from C,
which provides a mechanism similar to exception handling. therefore
most API functions can throw an error (that is, you
call longjmp) instead of returning a value.
When we write library code (that is, functions that will
can be called from Lua), using the longjmp function is almost like this
as convenient as using real exceptions because
Lua will catch any error that occurs. When we write code for-
definitions (that is, the C code that calls Lua), then we must
provide a way to catch such errors.

Error handling in application code

When your application calls functions from the Lua API, then it sub-
liable to mistakes. As we discussed, Lua usually reports
these errors using the longjmp function . However, if there is no corresponding

the corresponding call to setjmp , the interpreter cannot execute
and longjmp . In this case, any error in the API results in
Lua calls a *special function* (panic function), and if the control
is returned from this function, then the execution of the application will
rummages. You can define your similar function with
lua_atpanic , but there isn't much that function can do.
In order to properly handle errors in your code
application, you have to call your code through Lua, so it will
Sets the appropriate context for catching errors (i.e.
it will execute your code in the context of setjmp). Just like we can-
Let's run Lua code in protected mode using pcall ,
we can execute C code using lua_pcall . More accurately,
we put the C code in a function and call this function through
Lua using lua_pcall . (We will discuss in detail how to call
Lua C functions in Chapter 27.) Then our C code will run in
protected mode. Even in case of memory allocation error
lua_pcall returns the corresponding error code, leaving the
the interpreter is in working order.

Error handling in library code

Lua is a *safe* language. This means that it doesn't matter what you write
in Lua, no matter how wrong it is, you can always understand
running the program in terms of Lua itself. Moreover, errors are also
are discovered and explained in Lua terms. You can compare
this is with C, where the behavior of many mis-written programs can

can only be explained in terms of the equipment used
(for example, error locations are specified as command addresses).
When you add a C function to Lua, you break this
safety. For example, a function like poke that writes
an arbitrary byte at an arbitrary memory address, can add
This leads to a large number of errors when working with memory. You need to
make sure your additions are safe for Lua and
provided good error handling.
As we discussed earlier, C programs must specify
your error handling with lua_pcall . However, when you drink

you write arbitrary functions in Lua, usually you don't need to handle make mistakes. Errors thrown by the library function will be caught either by pcall in Lua or by lua_pcall in the application code. Therefore, when a function in a C library is catches an error, it might just call lua_error (or whatever even better is luaL_error , which formats the error message and then calls lua_error). The lua_error function clears everything that needs to be cleared in Lua, and jumps back to the protected output I call, passing the error message.

Exercises

Exercise 25.1. Compile and run a simple separate Lua interpreter (Listing 25.1).

Exercise 25.2. Let's assume the stack is empty. What will be on-walk on the stack after the next call sequence wwow?

```
lua_pushnumber (L, 3.5);
lua_pushstring (L, “hello”);
lua_pushnil (L);
lua_pushvalue (L, -2);
lua_remove (L, 1);
lua_insert (L, -2);
```

Exercise 25.3. Use the Lua interpreter from listing ha 25.1 and the stackDump function (Listing 25.2) in order to check your answer to the previous exercise.

Chapter 26
Extending your applications

An important use of Lua is to use it as a *configurative* language. In this chapter we will show how we can use use Lua to configure a program, starting with a simple example, and we will develop it to perform more and more complex tasks.

26.1. The basics

As the first task, let's look at a simple configuration
rational script: your C program has a window and you want
be able to set the initial window size. It is clear that for
such a simple problem, there are simpler solutions than
Lua uses such as environment variables or files with
name-value pairs. But, even using a simple text file,
you somehow need to disassemble it; so you decide to use
a Lua configuration file (i.e. a text file that
is a Lua program). In its simplest form, this text
the file may contain the following lines:

```
- define window size
width = 200
height = 300
```

You must now use the Lua API to enable Lua to parse
took this file, and then get the values of global variables
`width` and `height` . The `load` function in Listing 26.1 does this.
bot. This function assumes that you have already created the Lua state,
similar to what we saw in the previous chapter. She calls the function
clause `luaL_loadfile` to load the block from the file `fname` and then
binds `lua_pcall` to run compiled block. When
errors (for example, syntax errors in your config
on file), these functions push the error message onto the stack and
return a nonzero error code; our program then uses
is `lua_tostring` with index -1 in order to get the message
from the top of the stack. (We discussed the `error` function in Section 25.1.)

Listing 26.1. Getting user information
from the config file

```
void load (lua_State * L, const char * fname, int * w, int * h) {
if (luaL_loadfile (L, fname) || lua_pcall (L, 0, 0, 0))
error (L, “cannot run config. file:% s”, lua_tostring (L, -1));
lua_getglobal (L, “width”);
lua_getglobal (L, “height”);
if (! lua_isnumber (L, -2))
error (L, “'width' should be a number \ n”);
if (! lua_isnumber (L, -1))
error (L, “'height' should be a number \ n”);
* w = lua_tointeger (L, -2);
* h = lua_tointeger (L, -1);
}
```

After executing a block of code, the program needs to get the value global variables. To do this, she calls the function twice. ktsiyu lua_getglobal , whose parameter (except approach to others the accompanying lua_State) is the variable name. Everyone is like that the call pushes the corresponding value onto the stack, so the width the window will be at position with index –2 and height at position with index –1 (at the top). (Since the stack was initially empty, which we can also index starting from the bottom of the stack, then there is to use 1 for the first value and 2 for the second. But, indexing from the top, we do not need to make any assumptions the stack is empty .) Next, our example uses the lua_ isnumber for checking if each value is number. Then lua_tointeger is called , and the corresponding values values are assigned.

Was Lua worth using for a similar task? As I said earlier, for such a simple task, a simple text file with two lami will be much easier than Lua. Even so, using Lua gives us some advantages. First, Lua is fully concerned with syntax for you; your config file may even contain reap comments! Second, the user can already execute complex configuration with what we have. For example, the script can ask the user for some information or take value from the environment variable to select the appropriate size ra:

```
- configuration file
if getenv (“DISPLAY”) == “: 0.0” then
width = 300; height = 300
else
width = 200; height = 200
end
```

Even in such simple scenarios, configuration is difficult to advance anticipate what users might want; but until the script defines these two variables, your C program will work unchanged.

The final reason for using Lua is that it is now easy to add new configuration options to your program; this ease creates an approach that leads to much more flexible programs.

26.2. Working with tables

Let's take this approach: now we want to set the background color for window. We will assume that the given color consists of three numbers, each of the numbers is an RGB component. Usually in C these are la are integer in some range for example [0, 255]. In Lua, since all numbers are floating point, we will have more it is natural to use the range [0, 1].

A naive approach would be to ask the user to ask each component in a separate global variable:

```
- configuration file
width = 200
height = 300
background_red = 0.30
background_green = 0.10
background_blue = 0
```

This approach has two drawbacks: first, it is too redundant. and cumbersome (real programs may need dozens of colors for the background in the window, for the color in the window, for the background of the menu, etc.); and there is no way to pre-define common colors so that the user could then simply write `background = WHITE` . To avoid these disadvantages, we can represent colors by cabbage tables:

```
background = {r = 0.30, g = 0.10, b = 0}
```

Using tables gives structure to your script; Now easy for the user (or application) to define colors for further use in the config file:

```
BLUE = {r = 0, g = 0, b = 1.0}
<other color definitions>
background = BLUE
```

To obtain these values on C, we can do the following - in a way:

```
lua_getglobal (L, “background”);
if (! lua_istable (L, -1))
error (L, “'background' is not a table”);
red = getcolorfield (L, “r”);
green = getcolorfield (L, “g”);
blue = getcolorfield (L, “b”);
```

Listing 26.2. An example implementation of the getcolorfield function

```
#define MAX_COLOR 255
/ * assume that the table is on the top of the stack * /
```

```
int getcolorfield (lua_State * L, const char * key) {
int result;
lua_pushstring (L, key); / * push the key onto the stack * /
lua_gettable (L, -2); / * get background [key] * /
if (! lua_isnumber (L, -1))
error (L, “invalid component in background color”);
result = (int) (lua_tonumber (L, -1) * MAX_COLOR);
lua_pop (L, 1); / * remove number * /
return result;
}
```

We first get the value of the global variable back-
ground and make sure it is a table and then use get-
colorfield to get each component.

Of course the getcolorfield function is not part of the API, we have to define it. Again we are faced with the problem of polymorphism: there can be many versions of getcolorfield function that differ key type, value type, error handling, etc. Lua API offers just one lua_gettable function that works for of all types. She takes the position of the table on the stack, removes the key from stack and pushes the corresponding value onto the stack. Our function getcolorfield , defined in Listing 26.2, assumes that the table is is at the top of the stack, so after putting the key on the stack using the lua_pushstring function , the table will be located at index –2. Before returning, getcolorfield pops a semi-value, leaving the stack in the same state it was in before this call.

Since indexing a table with a string key cha is very common, Lua 5.1 introduced a specialized version lua_gettable is just for this case: lua_getfield . Using this function, we can rewrite the following two lines:

```
lua_pushstring (L, key);
lua_gettable (L, -2); / * get background [key] * /
```

as

```
lua_getfield (L, -1, key);
```

(Since we are not pushing a row onto the stack, the table has an index is still -1 when lua_getfield is called .)

We will expand our example a little and introduce names in color for user. The user can still use the tab-faces for the component color setting, but you can also use Call predefined color names. To implement this

we need a color table in our C program:

```
struct ColorTable {
char * name;
unsigned char red, green, blue;
} colortable [] = {
{“WHITE”, MAX_COLOR, MAX_COLOR, MAX_COLOR},
{“RED”, MAX_COLOR,
0,
0},
{“GREEN”,
0, MAX_COLOR,
0},
{“BLUE”,
0,
0, MAX_COLOR},
<other colors>
{NULL, 0, 0, 0} / * terminator * /
};
```

Our implementation will create global variables with color-coded names comrade and initializes these variables using color tables
Comrade The result will be the same if the user added
the following lines into your script:

```
WHITE = {r = 1.0, g = 1.0, b = 1.0}
RED = {r = 1.0, g = 0, b = 0}
<other colors>
```

To set the fields of the table, we will introduce a helper function setcolorfield ; it pushes the index and value of the field onto the stack and then calls lua_settable :

```
/ * consider the table to be at the top of the stack * /
void setcolorfield (lua_State * L, const char * index, int value) {
lua_pushstring (L, index); / * key * /
lua_pushnumber (L, (double) value / MAX_COLOR); / * value * /
lua_settable (L, -3);
}
```

Similar to other API functions, lua_settable works for many different types, so it pops all of its operands off the stack. It takes the index of the table as an argument and strips off the key and value from the stack. The setcolorfield function assumes that before calling the tab-face is at the top of the stack (index –1); after placing in-dex and values on the stack, the table will be at index –3.
Lua 5.1 also introduced a specialized version of lua_settable for string keys, it's called lua_setfield . Using this no-function, we can rewrite setcolorfield as follows

at once:

```
void setcolorfield (lua_State * L, const char * index, int value) {
lua_pushnumber (L, (double) value / MAX_COLOR);
lua_setfield (L, -2, index);
}
```

The next function, setcolor , defines one color. She created creates a table, sets the values of the corresponding fields and maps this table to the corresponding global variable:

```
void setcolor (lua_State * L, struct ColorTable * ct) {
lua_newtable (L); / * creates a table * /
setcolorfield (L, “r”, ct-> red); / * table.r = ct-> r * /
setcolorfield (L, “g”, ct-> green); / * table.g = ct-> g * /
setcolorfield (L, “b”, ct-> blue); / * table.b = ct-> b * /
lua_setglobal (L, ct-> name); / * 'name' = table * /
}
```

Lua_newtable function creates an empty table and places it on the stack; setcolorfield calls set the fields of this table; finally, lua_setglobal pops the table off the stack and uses it as a value global variable with the given name.

Using these functions, the next cycle will register all colors. the one for the config script:

```
int i = 0;
while (colortable [i] .name! = NULL)
setcolor (L, & colortable [i ++]);
```

Remember that the application must execute this loop before you filling in the script.

Listing 26.3. Colors as rows or tables

```
lua_getglobal (L, “background”);
if (lua_isstring (L, -1)) {/ * is the value a string? * /
const char * name = lua_tostring (L, -1); / * get string * /
int i; / * look in the table * /
for (i = 0; colortable [i] .name! = NULL; i ++) {
if (strcmp (colorname, colortable [i] .name) == 0)
break;
}
if (colortable [i] .name == NULL) / * string not found? * /
error (L, “invalid color name (% s)”, colorname);
else {/ * use colortable [i] * /
red = colortable [i] .red;
green = colortable [i] .green;
blue = colortable [i] .blue;
}
} else if (lua_istable (L, -1)) {
red = getcolorfield (L, “r”);
green = getcolorfield (L, “g”);
```

```
blue = getcolorfield (L, “b”);
} else
error (L, “invalid value for 'background'”);
```

Listing 26.3 shows another option for implementing name bathroom flowers. Instead of global variables, the user can can denote colors using strings, writing the settings as `background = ”BLUE”` . So `background` can be either table, or row. With this approach, the application does not need do something before running the script. Instead, to get colors have a little more work to do. When the program is receives the value of the `background` variable , then you need to check if is this value a string, in which case look for a color in color table.

What's the best option? In C programs, using lines to indicate options is not good practice since the compiler cannot detect typos. However, in Lua the message about a typo in the name of the color will go to the person for whom it is written this configuration. Difference between programmer and user somewhat blurry; difference between compilation error and error runtime is not that great.

With strings, the value of the `background` variable can be string Coy with a typo; in this case the application can add this information formation to the error message. The application can also compare thread strings regardless of the case of letters, so that the user can write `“white”` , `“WHITE”` or even `“White”` . Moreover, if the script is small and many errors are found, then this may be not very successful - add hundreds of colors (and create hundreds of tabs) persons and global variables) only for the user to chose several colors. With strings, you avoid this.

26.3. Lua function calls

The strength of Lua is that the configuration file can can define functions that can then be called by the application genius. For example, you can write an application to plot the function, and use Lua to plot give a function whose graph will be plotted.

The API provided way to call functions is pretty simple:
first, you push the function to be called onto the stack;
second, you put the arguments to be called on the stack as well; after that-
go use lua_pcall to call the function and finally remove
results from the stack.
Let, as an example, our config file contains
there is a function like the one below:

```
function f (x, y)
return (x ^ 2 * math.sin (y)) / (1 - x)
end
```

You want to compute z = f (x, y) in C for given x and y . Considering
that you have already opened the Lua library and performed the configuration-
file, the function f in Listing 26.4 implements this call.

Listing 26.4. Calling a Lua function from C

```
/ * call function 'f' defined in Lua * /
double f (lua_State * L, double x, double y) {
int isnum;
double z;
/ * push function and arguments onto the stack * /
lua_getglobal (L, “f”); / * called function * /
lua_pushnumber (L, x); / * push 1st argument onto the stack * /
lua_pushnumber (L, y); / * push 2nd argument onto the stack * /
/ * call the function (2 arguments, 1 result) * /
if (lua_pcall (L, 2, 1, 0)! = LUA_OK)
error (L, “error running function 'f':% s”,
lua_tostring (L, -1));
/ * get the result * /
z = lua_tonumberx (L, -1, & isnum);
if (! isnum)
error (L, “function 'f' must return a number”);
lua_pop (L, 1); / * pop the result off the stack * /
return z;
}
```

The second and third arguments to lua_pcall are, respectively,
the number of arguments you pass and the number of results that
you want to receive. The fourth argument is a function
tions for error handling; we will discuss this soon. As with
assignments in Lua, calling lua_pcall casts a valid
the number of resulting values to the number you specify; if necessary
placing on the stack walk ***nil*** 's or removing the extra values. Front
pushing the results onto the stack lua_pcall removes the function from the stack.
tion and its arguments. When a function returns multiple values,
then the first value is pushed onto the stack first; for example, if possible
three values rotate, then the first of them will have index –3,

and the last –1.
If an error occurs while executing lua_pcall
the lua_pcall function returns an error code; besides, she put
pops the error message onto the stack (but still pops
function and its arguments). However, before posting a message to
the lua_pcall stack calls the message handling function if it
has been asked. To set the message processing function, use
those are the last argument to the lua_pcall function . Zero means no
no processing function and the final message is the outcome
new error message. Otherwise, this argument must
be the index on the stack where the message handling function is located
niya. In such a case, the processing function should be placed
on the stack to the called function and its arguments.
For normal errors, the lua_pcall function returns the code
LUA_ERRRUN errors . Two special types of errors deserve
separate codes, because they never call a function
processing unit. The first type is memory allocation errors. For
For similar errors, lua_pcall always returns LUA_ERRMEM . Second
the type of errors is errors when executing the handler itself
communication. In this case, there is no point in calling the request again.
the message bot, so lua_pcall returns immediately
Correction with the LUA_ERRERR code . Lua 5.2 highlights a third type of error:
when the finalizer throws an error, lua_pcall returns
LUA_ERRGCMM code . This code indicates that the error is not related to
with the challenge itself.

Listing 26.5. Generalized function call

```
#include <stdarg.h>
void call_va (lua_State * L, const char * func,
const char * sig, ...) {
va_list vl;
int narg, nres; / * number of arguments and results * /
va_start (vl, sig);
lua_getglobal (L, func); / * push the function onto the stack * /
<put arguments on the stack (Listing 26.6)>
nres = strlen (sig); / * number of expected results * /
if (lua_pcall (L, narg, nres, 0)! = 0) / * make a call * /
error (L, “error calling '% s':% s”, func,
lua_tostring (L, -1));
<get the results (Listing 26.7)>
va_end (vl);
}
```

26.4. Generalized call function

As a more complex example, we will construct a universal
a function to call functions in Lua using vararg in C. Our
function, let's call it call_va , takes the name of the function that
need to be called, a string describing the types of arguments and results
tat, then a list of arguments, and finally a list of pointers to
the variables in which we want to get the results of the call. When
using this function, we can easily overwrite our previous
example as follows:

```
call_va (L, “f”, “dd> d”, x, y, & z);
```

The line “dd> d” means “two arguments of type double and one re-
The result is of type double . This specifier uses 'd' for double , 'i'
for integers and 's' for strings; the '>' character separates arguments from
results. If the function returns nothing, then the '>' symbol is not-
required.

Listing 26.5 shows the implementation of the call_va function . Despite
As for the general view of this function, it follows the same path as our
first example: pushes a function onto the stack, pushes arguments to
the stack (Listing 26.6), makes the call, and receives the results (Listing
26.7). Most of the code is pretty straightforward, but there are some subtle
ty. First, it doesn't check that func is a function; if a
it is not, then lua_pcall will raise an error. Secondly, since it
pushes an arbitrary number of arguments onto the stack, it must check
Check if there is free space on the stack. Thirdly, since the functional
tion can return rows, then call_va cannot remove results from
stack. This should be done by the caller after use.
result strings (or copy them elsewhere).

Listing 26.6. Generally pushing arguments

```
for (narg = 0; * sig; narg ++) {/ * execute for each argument * /
/ * check stack space * /
luaL_checkstack (L, 1, “too many arguments”);
switch (* sig ++) {
case 'd': / * double argument * /
lua_pushnumber (L, va_arg (vl, double));
break;
case 'i': / * int argument * /
```

```
lua_pushinteger (L, va_arg (vl, int));
break;
case 's': / * string argument * /
lua_pushstring (L, va_arg (vl, char *));
break;
case '>': / * end of arguments * /
goto endargs;
default:
error (L, “invalid option (% c)”, * (sig - 1));
}
}
endargs:
```

Listing 26.7. Getting call results

```
nres = -nres; / * index of the first result on the stack * /
while (* sig) {/ * repeat for each result * /
switch (* sig ++) {
case 'd': {/ * double result * /
int isnum;
double n = lua_tonumberx (L, nres, & isnum);
if (! isnum)
error (L, “wrong result type”);
* va_arg (vl, double *) = n;
break;
}
case 'i': {/ * int result * /
int isnum;
int n = lua_tointegerx (L, nres, & isnum);
if (! isnum)
error (L, “wrong result type”);
* va_arg (vl, int *) = n;
break;
}
case 's': {/ * string result * /
const char * s = lua_tostring (L, nres);
if (s == NULL)
error (L, “wrong result type”);
* va_arg (vl, const char **) = s;
break;
}
default:
error (L, “invalid option (% c)”, * (sig - 1));
}
nres ++;
}
```

Exercises

Exercise 26.1. Write a C program that reads
a Lua file that defines a function `f` that takes as input
number and returns the function value from this number. Your
the program should plot this function. (You do not-
be sure to use graphics, the usual one is fine
text view using `'*'` for graph.)

Exercise 26.2. Modify the `call_va` function (Listing 26.5)
to handle boolean values.

Exercise 26.3. Let there be a program that needs
keep track of several weather stations. Inside, for
representation of each station, it uses a 4-byte
line, and there is a config file that matches
Each such line contains the URL of the corresponding station. Con-
the Lua figure file must do this mapping.
in several different ways:

• a set of global variables, one for each stan-
tion;
• one table mapping strings to URL;
• one function that returns a URL for each line.

Discuss the pros and cons of each option, taking
attention total number of stations, types of users, availability
structures in the URL, etc.

Chapter 27
Calling C from Lua

When we say that Lua can call C, it doesn't mean that Lua
may call any function in C [1] . As we saw in the previous
chapter, when C calls a function in Lua, you must follow the
a specific protocol for passing arguments and receiving results
that. Similarly, in order for Lua to call a function in C, this function
must follow a specific protocol to get their
arguments and return results. Moreover, so that Lua can
call a function in C, we must register this function,
that is, they must pass Lua its address in a certain way.
When Lua calls a function in C, it uses the same
the stack that C uses to call Lua code. Function in C semi-
pops its arguments off the stack and pushes its results onto the stack.
The important concept here is that the stack is some
structure; each function has its own local stack.
When Lua calls a C function, the first argument will always be
have index 1 on this local stack. Even when the C code calls
Lua code that calls the same (or a different) function, each
of these calls will only see his personal stack with the first ar-
by index 1.

27.1. Functions in C

As a first example, let's see how to implement
a simplified version of a function that returns the sine of a given
numbers:

```
static int l_sin (lua_State * L) {
double d = lua_tonumber (L, 1); / * get argument * /
lua_pushnumber (L, sin (d)); / * push the result onto the stack * /
```

[1] There are packages that allow Lua to call any function in C, but they don't.

portable and not secure.

```
return 1; / * number of results * /
}
```

Any function registered in Lua must have one and
the same prototype defined in lua.h file as lua_CFunction :

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

From the point of view of C, the function on C receives as its unique
argument is the Lua state and returns an integer equal to
the number of values returned through the stack. Therefore, the functions do not need
but clear the stack before pushing your results onto it. Pos-
lua itself saves the results and cleans up
stack.
Before we can use this feature, we must
us first to register it. We do this with lua_
pushcfunction : it receives a pointer to a C function and creates
a value of type "function" that represents this function internally
ri Lua. After registration, the C function behaves like any other
function inside Lua.
A quick and dirty way to check the l_sin function is
put its code directly into our base interpreter
(Listing 25.1) and add the following lines right after the call

```
luaL_openlibs :
lua_pushcfunction (L, l_sin);
lua_setglobal (L, "mysin");
```

The first line pushes the function type value onto the stack, and the second
assigns its value to the global variable mysin . After these
changes, you can use this new function directly in your
their Lua scripts. (In the next section we will look at more correct
good ways to connect C functions to Lua.)
For a more serious sine function, we must
check the type of its argument. Here we are helped by an auxiliary
library. LuaL_checknumber function checks if
whether the given argument is a number: if an error occurs, it is thrown
It has a meaningful error message, otherwise it returns the number itself.
The changes to our function are minimal:

```
static int l_sin (lua_State * L) {
double d = luaL_checknumber (L, 1);
lua_pushnumber (L, sin (d));
return 1; / * number of results * /
}
```

With this function definition, if we call mysin ('a') , then
we will receive the following message:

```
bad argument # 1 to 'mysin' (number expected, got string)
```

Notice how luaL_checknumber is automatically filled in
reads the message by argument number (# 1), function name (“mysin”),
expected parameter type (number) and real parameter type
(string).

As a more complex example, let's write a function,
which will return the contents of the given directory. Lua is not provided
has such a function in its standard libraries, because
ANSI C does not have a suitable function for this. Here we will consider
that our function supports POSIX. Our function - let's call
its dir in Lua and l_dir in C - takes as an argument a string with
path to a directory and returns an array with the contents of this directory.
For example, calling dir (“/ home / lua”) might return the following table
person {“.”, ”..”, ”src”, ”bin”, ”lib”} . In case of error, the function
returns ***nil*** and an error string. The complete code of this
the function is shown in Listing 27.1. Pay attention to the use
calling the luaL_checkstring function , which behaves similarly
luaL_checknumber , but only for strings.

(In extreme cases, using this function may result in
to a small memory leak. Three of the Lua functions it calls are
may fail due to insufficient memory
ty: lua_newtable , lua_pushstring and lua_settable . If any of
these functions will fail, then an error will be raised and
complements l_dir be interrupted, respectively, closedir will not
caused. As we discussed earlier, for most programs this is not
is a big problem: if memory runs out, then the best
what can be done is to terminate the execution of the program. However
less in chapter 30 we will see another implementation of the getter function
contents of the directory, which no longer contains this error.)

Listing 27.1. Function for reading directory contents

```
#include <dirent.h>
#include <errno.h>
#include <string.h>
```

```
#include “lua.h”
#include "lauxlib.h"
static int l_dir (lua_State * L) {
DIR * dir;
struct dirent * entry;
int i;
const char * path = luaL_checkstring (L, 1);
/ * open directory * /
dir = opendir (path);
if (dir == NULL) {/ * error opening directory? * /
lua_pushnil (L); / * return nil ... * /
lua_pushstring (L, strerror (errno)); / * and message * /
return 2; / * number of results * /
}
/ * create a table with the result * /
lua_newtable (L);
i = 1;
while ((entry = readdir (dir))! = NULL) {
lua_pushnumber (L, i ++); / * push key * /
lua_pushstring (L, entry-> d_name); / * push value * /
lua_settable (L, -3);
}
closedir (dir);
return 1; / * table is already at the top of the stack * /
}
```

27.2. Continuations

With lua_pcall and lua_call , a C function called from Lua can, in turn, call Lua. Some functions from the standard Noah libraries do this: table.sort can call a function comparisons; string.gsub can call the replace function; pcall and xpcall can call functions in protected mode. If we help- him that the main Lua code was itself, in turn, called from C (main program), then we get a sequence like next: C (application) calls Lua (script), which calls gives a C function (library) that calls Lua.
Lua normally handles these call sequences without problems; after all, its main task is to integrate with C. However, there is a situation in which such a chain of calls can cause problems: coroutines.
Every coroutine in Lua has its own stack, which contains information about pending coroutine calls. More precisely,

the stack remembers the return address, parameters and local variables
every call. For function calls in Lua, the interpreter is used
calls a suitable data structure to implement the stack, which is
It is called *a flexible stack* (soft stack). However, for function calls in C,
The interpreter must also use the C stack. After all, the address
return and local variables of a function in C live on the C stack.
An interpreter can easily have many flexible stacks, but the code for
C only has one stack. Therefore, coroutines in Lua cannot
stop execution inside the function in C: if in the call chain
from resume to the corresponding yield there is a function in C, then Lua
cannot save the state of this function in order to restore
renew it the next time you resume . Let's consider the next
example in Lua 5.1:

```
co = coroutine.wrap (function (a)
return pcall (function (x)
coroutine.yield (x [1])
return x [3]
end, a)
end)
print (co ({10, 3, -8, 15}))
-> false attempt to yield across metamethod / C-call boundary
```

The pcall call is a C function; therefore Lua cannot “freeze
zit "it, because there is no ANSI C way to pause execution
function in C and then continue its execution.
Lua 5.2 has dealt with this complexity with *continuations*
(continuation). Lua 5.2 implements yield with longjmp , i.e.
the same way it implements errors. Such a call (longjmp) is simply from-
discards all information about functions on the C-stack, so there is no
you can continue the function in C. However, the function in C foo can
specify a function - continuation of foo-c that is different
a C function to be called when the time is right
"Continue" function foo . That is, the interpreter detects
that it should continue executing foo , but since all the information
mation about the foo was destroyed with the stack, instead of that, he is
foo-c .
To make it clearer, let's look at an example:
implementation of the pcall function . In Lua 5.1. this function had the following-
the following code:

```
static int luaB_pcall (lua_State * L) {
int status;
luaL_checkany (L, 1); / * at least one parameter * /
```

```
status = lua_pcall (L, lua_gettop (L) - 1, LUA_MULTRET, 0);
lua_pushboolean (L, (status == 0)); / * status * /
lua_insert (L, 1); / * status is the first result * /
return lua_gettop (L); / * return status + all results * /
}
```

If the function called via lua_pcall is suspended
if it was executed (via yield), it will be impossible to continue
live later execution of luaB_pcall . Therefore, the interpreter will issue
error every time we try to call yield inside the
a protected call. Lua 5.2 implements pcall like
shown in Listing 27.2 [2] . There are three differences from the Lua 5.1 version:
firstly, the new version replaced the lua_pcall call with lua_pcallk ;
secondly, she has grouped everything that is done after this call in
the new finishpcall helper function ; the third difference is
the pcallcont function , the last argument is lua_pcallk , which is
is a continuation function.

If there are no yield calls , then lua_pcallk works exactly like
lua_pcall . If there is a call to yield, then everything is completely different. If a
a function called by lua_pcall tries to call yield , then Lua 5.2
raises an error, just like Lua 5.1. However, when the function called
lua_pcallk , calls yield , then there is no error: Lua calls
longjmp and discards the entry for luaB_pcall from the C stack, but keeps
This adds a reference to the pcallcont continuation function in the flexible stack .
Later, when the interpreter finds that it should return
to luaB_pcall (which is not possible), it calls the function instead-
continued pcallcont .

Unlike luaB_pcall , pcallcont 's continuation function cannot
can get the value returned by the lua_pcallk call . therefore
Lua provides a special function to return status
call: lua_getctx . When called from a normal Lua function (which
in our case does not happen), lua_getctx returns LUA_OK . Kog-
yes it is called from a continuation function, it returns LUA_YIELD .
The continuation function can also be called in the case of some
errors; in this case lua_getctx returns an error code, that is
the very value that lua_callk would return in this case .

Listing 27.2. A pcall implementation with continuations

```
static int finishpcall (lua_State * L, int status) {
lua_pushboolean (L, status); / * first result (status) * /
lua_insert (L, 1); / * put the first result in the first slot * /
return lua_gettop (L);
}
```

```
static int pcallcont (lua_State * L) {
int status = lua_getctx (L, NULL);
return finishpcall (L, (status == LUA_YIELD));
}
```
2
The actual code is more complex than shown here, since it has some general
other parts with xpcall and checking for stack overflow before placing on it
boolean value.
```
static int luaB_pcall (lua_State * L) {
int status;
luaL_checkany (L, 1);
status = lua_pcallk (L, lua_gettop (L) - 2, LUA_MULTRET, 0,
0, pcallcont);
return finishpcall (L, (status == LUA_OK));
}
```
In addition to the call status, the lua_getctx function can also return
context information. The fifth parameter in lua_pcallk is a
a free integer that can be obtained through the second parameter
lua_getctx , which is a pointer to an integer value. This is the price
the left value allows the original function to pass an arbitrary
information directly to its continuation. She can also convey
more information via the Lua stack. (Our example is not used
takes this opportunity.)
The continuation mechanism in Lua 5.2 is awesome
to support yield , but this is not a panacea. Some C function
may need to convey too much context to your pro
posts. You can use table as such an example .
sort , which uses the C stack for recursion, and string.gsub , which
paradise should keep track of found substrings and a buffer for partial
results. Although they can be rewritten in a way that supports
yuschim yield statement , the gain is not worth the complexity introduced.

27.3. C modules

A module in Lua is a block of code that defines various functions.
tions in Lua and remembers them in suitable places, usually tab-
faces. A C module for Lua behaves the same way. Besides defining
its functions in C, it must also define a special function
tion, which acts as the main block in the Lua library. This
the function must register all C functions from the module and
remember them in appropriate places, usually in table fields.

Like the main block in Lua, this function also needs to
to initialize everything that needs initialization.
Lua obtains functions in C through the registration mechanism. After
Once a C function is represented and stored in Lua, Lua calls
her through a direct link to her address (which we transmit
Lua when we register this function). In other words, Lua is not
depends on the function name, package location, or visibility rules
ty to call this function when it is registered.
Usually a C module has only one extern function,
which is the function that opens the library. Everything else-
functions can be closed, for example, by declaring them as
static .
When you extend Lua with C functions, a good
the idea is to organize your code as a C module, even if
you only want to register one function: sooner or later
(usually early) you will need other features. As usual, help
The powerful library offers a helper function for
this. The luaL_newlib macro takes a list of C functions along with their
appropriate names and registers them all inside the new tab-
faces. As an example, let's say we want to create a library with a function
tion l_dir , which we defined earlier. First, we must
define library functions:

```
static int l_dir (lua_State * L) {
<as before>
}
```

Next, we define an array with all the functions along with their
names. This array contains elements of type luaL_Reg , which
is a structure of two fields: the function name (string) and the
a function factor.

```
static const struct luaL_Reg mylib [] = {
{“Dir”, l_dir},
{NULL, NULL} / * terminator * /
};
```

In our example, there is only one function (l_dir), which we
want to register. The last pair in the array is always
{NULL, NULL} denoting the end of the array. Finally, we define
We write the main function using luaL_newlib :

```
int luaopen_mylib (lua_State * L) {
luaL_newlib (L, mylib);
return 1;
}
```

The luaL_newlib call creates a new table and fills it in pairs
name is a function from the mylib array . Upon returning luaL_newlib OS-
Pushes a new table onto the stack. Function luaopen_mylib vozvrascha-
is 1 in order to return this table to Lua.
After completing the library, we must link it to
interpreter. The easiest way to do this is
using dynamic libraries if the interpreter is Lua
supports them. In this case, you must create a dynamic
the library with your code (mylib.dll on Windows and mylib.so on Linux)
and place it along the C-path. After these steps, you can download
this library directly from Lua using require :

```
local mylib = require “mylib”
```

This call loads a dynamic library in Lua, find
dumps the luaopen_mylib function , registers it as a C function, and
calls it, thereby opening the module. (This explains why
luaopen_mylib must have the same prototype as any
another function in C.)
When loading a dynamic library, we need to know the name
function luaopen_mylib in order to find it. Will always be used
a function named luaopen_ , to which the name is attached
module. Therefore, if your module is called mylib , then the function
should be named luaopen_mylib .
If your interpreter does not support dynamic linking-
ku, then you need to rebuild Lua along with your new library.
Besides this rebuilding, you also need some way of saying
to the interpreter that it should open this library when creating
new state. This is usually done by adding luopen_mylib to
list of standard libraries that luaL_openlib opens in
file linit.c .

Exercises

Exercise 27.1. Write a summation function in C that
calculates the sum of a variable number of its numeric arguments:

```
print (summation ())
-> 0
print (summation (2.3, 5.4))
-> 7.7
print (summation (2.3, 5.4, -34)) -> -26.3
```

print (summation (2.3, 5.4, {}))
-> stdin: 1: bad argument # 3 to 'summation'
(number expected, got table)

Exercise 27.2. Implement a function equivalent to table.
pack from the standard library.

Exercise 27.3. Write a function that gets produced
free number of parameters and returns them in reverse order:
print (reverse (1, “hello”, 20)) -> 20 hello 1

Exercise 27.4. Write a foreach function that gets-
takes a table and a function as input and calls this function to
each key-value pair in the table:
foreach ({x = 10, y = 20}, print)
-> x 10
-> y 20

Exercise 27.5. Rewrite the foreach function from the previous
th exercise so that the called function can call
yield .

Exercise 27.6. Create a C module with all the functions from
previous exercises.
Exercises

Chapter 28
Writing techniques functions in C

Both the official API and the *auxiliary* library provide
provide several mechanisms to help write functions in C.
In this chapter, we will look at mechanisms for working with arrays,
strings and storing Lua values in C.

28.1. Working with arrays

In Lua, "array" is just a name for the table used by the spec.
in a special way. We can work with arrays using those
the same functions that we used to work with the table-
mi, that is, lua_settable and lua_gettable . However, the API is provided by
There are several special functions for working with arrays. One
one of the advantages of using these functions is performance:
often we have access to an array inside the loop of the algorithm

ma (for example, sorting), so that any increase in speed
the viya in these operations can have a big impact
on the final performance of the algorithm. Another plus is
convenience, integer keys are common enough that-
would deserve special treatment.
The API provides two functions for working with arrays:

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

The description of the lua_rawgeti and lua_rawseti functions is somewhat confusing.
is safe, since it includes two indices at once: index describes where
the table is on the stack; key specifies an element in the table itself.
The lua_rawgeti (L, t, key) call is equivalent to the following sequence
when t is greater than zero (otherwise it is necessary
compensate for the appearance of a new element on the stack):

```
lua_pushnumber (L, key);
lua_rawget (L, t);
```

Calling lua_rawseti (L, t, key) (again for positive t) eq-
is equivalent to the following sequence:

```
lua_pushnumber (L, key);
lua_insert (L, -2); / * put 'key' below the previous value * /
lua_rawset (L, t);
```

Note that both functions use direct
to the table. They are faster and moreover the tables used
as arrays, metamethods are rarely used.

Listing 28.1. C map function

```
int l_map (lua_State * L) {
int i, n;
/ * 1st argument must be table (t) * /
luaL_checktype (L, 1, LUA_TTABLE);
/ * 2nd argument must be function (f) * /
luaL_checktype (L, 2, LUA_TFUNCTION);
n = luaL_len (L, 1); / * get table size * /
for (i = 1; i <= n; i ++) {
lua_pushvalue (L, 2); / * push f * /
lua_rawgeti (L, 1, i); / * push t [i] onto the stack * /
lua_call (L, 1, 1); / * call f (t [i]) * /
lua_rawseti (L, 1, i); / * t [i] = result * /
}
return 0; /* No results */
}
```

As an example of using these functions, Listing 28.1 re-
lizuet function map : it applies a given function to all element
the array, replacing each element with the result of the call. This
the example also introduces three new functions: luaL_checktype , luaL_len

and lua_pcall .
The luaL_checktype function (from the lauxlib.h file) checks that
the given argument is of the given type, otherwise it is
throws an error.
The lua_len primitive (not used in the example above) is equivalent to
is valent to the '#' operator . Due to metamethods, this operator can
return an object of any type, not just numbers; so lua_len
returns its result on the stack. LuaL_len function (using
bathroom in the example) raises an error if the length is not pure
scrap, otherwise it returns the length as normal
integer type.
The lua_call function makes an unsecured call. He is analogous
lua_pcall is hygienic , but it passes the errors above rather than returning the code
errors. When you write the main application code, you better not
use lua_call as you want to catch any errors.
However, when you write functions, it is better to use exactly
lua_call ; if an error occurs, then we will leave it for someone,
to whom it is important.

28.2. Working with strings

When a C function receives a string argument from Lua, there is
there are only two rules that need to be followed: do not remove
string off the stack when working with it and never modify the string.
The situation gets more complicated when C functions need to be created
string to return it to Lua. C code should worry
about buffer allocation / deallocation, buffer overflow, etc.
at least the Lua API provides several functions for this.
The Standard API provides assistance in two of the most common
common operations: substring extraction and concatenation
lines. When you select a substring remember that lua_pushlstring po-
takes the length of the string as an optional argument. So if you
want to pass Lua substring of string s with characters at positions from i
up to j (inclusive), then all you have to do is:

```
lua_pushlstring (L, s + i, j - i + 1);
```

As an example, let's say you want a function that breaks down
emits a string at a given delimiter (one character) and returns
Creates a table with substrings. For example, calling split (“hi: ho:

there ”,”: ”) should return the table {“ hi ”,” ho ”,” there ”} . Fox
Thing 28.2 shows a simple implementation of this function. She doesn't need
we have additional buffers and it does not impose any restrictions
on the size of the lines that it can handle. About all
the buffers are taken care of by Lua itself.

Listing 28.2. Splitting a string

```
static int l_split (lua_State * L) {
const char * s = luaL_checkstring (L, 1); /* line */
const char * sep = luaL_checkstring (L, 2); /* delimiter */
const char * e;
int i = 1;
lua_newtable (L); / * table with result * /
/ * repeat for each separator * /
while ((e = strchr (s, * sep))! = NULL) {
lua_pushlstring (L, s, es); / * substring per stack * /
lua_rawseti (L, -2, i ++); / * insert into table * /
s = e + 1; / * go beyond separator * /
}
/ * insert last substring * /
lua_pushstring (L, s);
lua_rawseti (L, -2, i);
return 1; / * return table * /
}
```

For string concatenation, Lua provides a special function
in its API called lua_concat . It is equivalent to opera-
torus of concatenation .. in Lua; it converts numbers to strings and when
calls metamethods if necessary. Moreover, she can immediately
concatenate more than two lines. Calling lua_concat (L, n) concatenate
pops (popping from the stack) n values and puts the result on top
stack.
Another useful function is lua_pushfstring :

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

It is somewhat similar to the sprintf function in that it creates a string
ku by format string and additional arguments. However, in excellent
from sprintf , you don't need to provide a buffer. Lua dynamically
creates a string for you as large as needed. This function
pushes the resulting string onto the stack and returns a pointer to
her. You don't need to worry about buffer overflows.
Currently, this function only supports the following 1 formats :

% s
Insert null terminated string
% d
Insert integer
% f

Insert Lua number i.e. double
% p
Insert Pointer
% c
Insert integer as character
%%
Insert character '%'
No modifiers supported such as width
or accuracy.
Both lua_concat and lua_pushfstring are useful when we want to
concatenate only a few lines. However, if we need to connect
many lines (or characters) together, then do it one at a time
can be quite inefficient, as we saw in section 11.6.
In this case, we can use the buffers provided by
auxiliary library.
In the simplest case, buffers work like two functions: one
gives you a buffer of any size where you can write your string;
the other converts the buffer to a string in Lua 2 . Listing 28.3 showing
Can't use these functions by implementing the function
string.upper directly from the original lstrlib.c file . The first step
using the buffer from the auxiliary library is
the phenomenon of a variable of type luaL_Buffer . The next step is
call luaL_buffinitsize to get a pointer to a buffer with a given
new size; then you can use this buffer to create-
his line. The last step is to call luaL_pushresultsize
to convert the contents of the buffer to a new Lua string on ver-
the stack bus. The size in this call is the final size of the string.
(Often, as in our example, this size is equal to the size of the buffer, but it
maybe less. If you do not know the exact size of the resulting
string, but you have its maximum size, then you can order
larger buffer.)

Listing 28.3. String.upper function

```
static int str_upper (lua_State * L) {
size_t l;
size_t i;
luaL_Buffer b;
const char * s = luaL_checklstring (L, 1, & l);
char * p = luaL_buffinitsize (L, & b, l);
for (i = 0; i <l; i ++)
p [i] = toupper (uchar (s [i]));
luaL_pushresultsize (& b, l);
return 1;
```

```
}
```

Note that the luaL_pushresultsize function is not
takes the Lua state as its first argument. After
the initialization buffer stores a reference to the state, so we don't
you need to pass it when calling other functions to work with
buffers.
We can also use these buffers without knowing the maximum
the length of the resulting string. Listing 28.4 shows a simplified re-
lisation of the table.concat function . In this function, we first call
Vai luaL_buffinit to initialize the buffer. Then we add

to the buffer elements one by one, in this case using the function
luaL_addvalue . Finally, luaL_pushresult releases the buffer and
Places the summary line at the top of the stack.

Listing 28.4. Simplified implementation of table.concat

```
static int tconcat (lua_State * L) {
luaL_Buffer b;
int i, n;
luaL_checktype (L, 1, LUA_TTABLE);
n = luaL_len (L, 1);
luaL_buffinit (L, & b);
for (i = 1; i <= n; i ++) {
lua_rawgeti (L, 1, i); / * get row from table * /
luaL_addvalue (b); / * add it to the buffer * /
}
luaL_pushresult (& b);
return 1;
}
```

The helper library provides several functions
to add values to the buffer: function luaL_addvalue to-
Adds the Lua string that is at the top of the stack; function
luaL_addlstring adds strings with the specified length; function
luaL_addstring adds a null terminated string and
the luaL_addchar function adds single characters. These functions
tions have the following prototypes:

```
void luaL_buffinit (lua_State * L, luaL_Buffer * B);
void luaL_addvalue (luaL_Buffer * B);
void luaL_addlstring (luaL_Buffer * B, const char * s, size_t l);
void luaL_addstring (luaL_Buffer * B, const char * s);
void luaL_addchar (luaL_Buffer * B, char c);
void luaL_pushresult (luaL_Buffer * B);
```

When you use a buffer, pay attention to the following.

After initializing the buffer, it stores some auxiliary
results on the Lua stack. Therefore, you cannot assume
that the top of the stack will remain where it was before you put
use a buffer. You can use the stack for others
tasks while working with a buffer, the main thing is that calls to push and pop
were balanced every time you use the buffer. Ex-
the key to this rule is the luaL_addvalue function , which
paradise assumes that the line to be added to the buffer was
pushed to the top of the stack.

28.3. Saving statein C functions

Often C functions need to store some kind of non-local data,
that is, data that will survive the current function call. In C we
usually we use global (extern) or static variables-
for this purpose. However, when you write library functions
for Lua, then using global or static variables
not a good solution. First, you cannot save
an arbitrary Lua value in a C variable. Secondly, the library,
which uses such variables will not work correctly with
multiple Lua states.
A Lua function has two basic places where you can store non-lightweight
local data: global variables and non-local variables.
The C API also provides two basic storage locations for non-local
data: registry and values associated with the function (upvalue).
The registry is a global table that can be accessed
only with a C [3] code . Typically the registry is used to
storage of data that will be used by several
modules. If you need to save data only for your mo-
dul or functions, then you must use the values associated
with function.

Registry

The registry is usually located at a *pseudo-index* , the value of which is defined
divided as LUA_REGISTRYINDEX . The pseudo-index looks like an index

on the stack, except that the associated values are not
walk on the stack. Most of the functions in the Lua API that accept
take indices as arguments, they also accept pseudoin-
dexes - except for those functions that change the stack, such
like lua_remove and lua_insert . For example, in order to get
the value associated with the "Key" in the registry, we can use
make the following call:

```
lua_getfield (L, LUA_REGISTRYINDEX, “Key”);
```

The registry is just a regular Lua table. Accordingly, you can for
to call it, use any Lua value other than ***nil*** . One-

[3] In fact, the registry can also be accessed from Lua using a function from the debugger.
in the debug.getregistry sub-library.

but since all C modules share the same registry, you
must be very careful in choosing the values that you will use
use as keys in order to avoid possible conflicts
Comrade String keys are especially handy when you want to allow
other modules access your data, since everything they
need is the name. There is no completely reliable method for
bunch of keys, but there are some proven approaches,
such as not using common names and starting your
names with the library name or something like it. (Prefixes like
lua and lualib are not good options.)
You should never use numbers as keys
registry, since such keys are reserved for the *system*
links (reference system). This system consists of a pair of functions in
helper libraries that allow you to save
readings in the table without worrying about the uniqueness of the keys.
Function
luaL_ref creates new links:

```
int r = luaL_ref (L, LUA_REGISTRYINDEX);
```

This call will pop the value from the top of the stack, store it in a table
face at the new integer index and will return that index. By-
Daubney indexes are called *links* (reference).
As the name suggests, we will use links mainly for
when we need to store a Lua value inside a struct
ry C. As we have already seen, we should never memorize a pointer
whether to Lua strings outside of the C function that received them. More
Moreover, Lua doesn't even offer pointers to other objects such as
tables or functions. Therefore, we cannot refer to objects

Lua with pointers. Instead, when we need that-
these are pointers, we will create links and store them in C.
To put the value associated with the reference r to
stack, we just use the following piece of code:
lua_rawgeti (L, LUA_REGISTRYINDEX, r);
Finally, in order to release both meaning and reference, we
call luaL_unref :
luaL_unref (L, LUA_REGISTRYINDEX, r);
After that, a new call to luaL_ref can return the same
link.
The referencing system treats ***nil*** as a special case. When we call
Vai luaL_ref for the value of ***nil*** , a new link is created, and instead
this returns the constant LUA_REFNIL . The next call to sa-
Actually does nothing:
luaL_unref (L, LUA_REGISTRYINDEX, LUA_REFNIL);
The next call pushes ***nil*** onto the stack , as expected:
lua_rawgeti (L, LUA_REGISTRYINDEX, LUA_REFNIL);
The reference system also defines the constant LUA_NOREF, which
is an integer other than any link. She is useful
in order to mark links as destroyed / uninitialized-
dyed.
Another reliable method for generating registry keys is
using a static variable as a key address in your
In the same code: the linker guarantees that this address is unique.
In order to use this option, you need the function
lua_pushlightuserdata , which pushes a value onto the Lua stack,
which is a pointer to C. The following code shows
how to save and retrieve a string from the registry using this method
Yes:
```
/ * variable with unique address * /
static char Key = 'k';
/ * remember the line * /
lua_pushlightuserdata (L, (void *) & Key); / * push address * /
lua_pushstring (L, myStr); / * push value * /
lua_settable (L, LUA_REGISTRYINDEX); / * registry [& Key] = myStr * /
/ * get string * /
lua_pushlightuserdata (L, (void *) & Key); / * push address * /
lua_gettable (L, LUA_REGISTRYINDEX); / * get value * /
myStr = lua_tostring (L, -1); / * convert it to string * /
```
We will discuss in more detail the use of the userdata type in
case 29.5.
In order to simplify the use of variable addresses in the

as unique keys, Lua 5.2 introduces two new functions: lua_
rawgetp and lua_rawsetp . They are like lua_rawgeti / lua_rawseti ,
but instead of integers, they use pointers (translated to
userdata) as keys. Using them, we can rewrite
the previous code is as follows:

```
static char Key = 'k';
/ * remember the line * /
lua_pushstring (L, myStr);
lua_rawsetp (L, LUA_REGISTRYINDEX, (void *) & Key);
/ * get string * /
lua_rawgetp (L, LUA_REGISTRYINDEX, (void *) & Key);
myStr = lua_tostring (L, -1);
```

Function related values

While the registry offers global variables, the mechanism
values associated with functions offers an analogue of static-variables-
in C, which are visible only inside a separate function. Each
time when you create a new C function in Lua you can bind
with it any number of similar values; every such value
is a Lua value. Then, when the function is called, it
accesses any of these values freely using
pseudo-indices.

We call this relationship functions in C with their values *za-
closures* (closure). The C closure is analogous to the Lua closure
for the C language. In particular, you can create various closures-
using the same function code, but different associated values
cheniya.

As a simple example, let's write a function
newCounter in C [4] . This function is a factory: it returns
Creates a new counting function each time it is called. Although everyone is like that
functions have the same C code, each of them stores its own
own counter. This factory function looks like this
way:

```
static int counter (lua_State * L); / * forward declaration * /
int newCounter (lua_State * L) {
lua_pushinteger (L, 0);
lua_pushcclosure (L, & counter, 1);
return 1;
```

```
}
```

The main function here is lua_pushcclosure , which co-
creates a new closure. Its second argument is the base function.
ration (in the example this is counter), and the third argument is a number
associated values (in the example, this is 1). Before creating a new job
we have to put the initial values for the related
values on the stack. In our example, we put 0 as the initial
value for the only associated value. As expected,
lua_pushcclosure leaves a new closure on the stack, so
the closure is already ready to return as the result of newCounter .
Let's now take a look at the definition of the counter function :

```
static int counter (lua_State * L) {
int val = lua_tointeger (L, lua_upvalueindex (1));
lua_pushinteger (L, ++ val); / * new value * /
lua_pushvalue (L, -1); / * duplicate it * /
lua_replace (L, lua_upvalueindex (1)); / * update value * /
return 1; / * return new value * /
}
```

The key element here is the lua_upvalueindex macro ,
which returns the pseudo-index of the associated value. In part
Nost expression lua_upvalueindex (1) returns the pseudo
the first associated value of the current function. This pseudo-index
looks like any index on the stack, only it's not on the stack.
Therefore, the call to lua_tointeger returns the current value first
th (and only) associated value as a number. Then the function
pushes the new value ++ val onto the stack, makes a copy of it, and
uses one of the copies to replace the associated value
nie. Finally, it returns another copy as its value.
As a more complex example, we will implement tuples
using associated values. *A tuple* is something like a constant
Yanny records with anonymous fields; you can get specific
new field by index or you can get all fields at once. In our
implementations, we will represent tuples as functions that
remember their values in related values. When the function
called with a numeric argument, then it returns the specific field.
When called with no arguments, it returns all fields. Next
The following code demonstrates the use of tuples:

```
x = tuple.new (10, “hi”, {}, 3)
print (x (1)) -> 10
```

```
print (x (2)) -> hi
print (x ()) -> 10 hi table: 0x8087878 3
```

In C, we represent all tuples using the same the t_tuple function , shown in Listing 28.5. Since we can let's call a tuple with a numeric argument or without arguments, the t_tuple function uses luaL_optint to reading an optional argument. LuaL_optint function is like luaL_checkint , but if the argument is absent, then it simply returns the specified default value (in the example it is 0).

Listing 28.5. Implementing tuples

```
int t_tuple (lua_State * L) {
int op = luaL_optint (L, 1, 0);
if (op == 0) {/ * no arguments? * /
int i;
/ * push each associated value onto the stack * /
for (i = 1;! lua_isnone (L, lua_upvalueindex (i)); i ++)
lua_pushvalue (L, lua_upvalueindex (i));
return i - 1; / * number of values on the stack * /
}
else {/ * get the 'op' field * /
luaL_argcheck (L, 0 <op, 1, “index out of range”);
if (lua_isnone (L, lua_upvalueindex (op)))
return 0; / * no field * /
lua_pushvalue (L, lua_upvalueindex (op));
return 1;
}
}
int t_new (lua_State * L) {
lua_pushcclosure (L, t_tuple, lua_gettop (L));
return 1;
}
static const struct luaL_Reg tuplelib [] = {
{“New”, t_new},
{NULL, NULL}
};
int luaopen_tuple (lua_State * L) {
luaL_newlib (L, tuplelib);
return 1;
}
```

When we refer to a non-existent associated value, the result is a pseudo-value of type LUA_TNONE . (When we accessing the value above the top of the stack, then we also get pseudo-value of type LUA_TNONE .) Therefore, our function t_tuple uses lua_isnone to check if there is a corresponding value

reading. However, we should never call lua_upvalueindex
with a negative index, so we must check this when
yes the index is provided by the user. LuaL_argcheck function
checks for any given value, throwing an error if needed
availability.
Function for creating t_new tuples (also in Listing 28.5)
trivial: since all of her arguments are already on the stack, she just
binds lua_pushcclosure to create a circuit using their
arguments as bound values. Finally, the tubplelib array and
the luaopen_tuple function (also in Listing 28.5) are standard
code to create a tuple library with a single function
new .

The values associated with the function are used by multiple functions

Quite often we need to give access to multiple values or
variables to all functions of this module. Although we can use
call the registry for this purpose, we can also use values
related to functions.
Unlike Lua closures, C closures cannot be shared.
associated values. Each closure has its own independent
related values. However, we can make it so that
values of several functions will point to the same
table, so this table becomes the environment where everything
these functions can store general data.
Lua 5.2 has a feature that makes the task of separating connections easier.
value between all library functions. We are open
Get C libraries with luaL_newlib . Lua implements this
function with the following macro:

```
#define luaL_newlib (L, l) \
(luaL_newlibtable (L, l), luaL_setfuncs (L, l, 0))
```

The luaL_newlibtable macro simply creates a table for the library.
theca. (We could have used lua_newtable as well , but this macro
uses lua_createtable to create a table with pre-emitted
small size, optimal for the number of functions in this library
lioteke.) Function luaL_setfuncs adds features from the list l to
a new table at the top of the stack.

We are interested in the third parameter of the luaL_setfuncs function here .
It tells how many related values the functions will have.
libraries. Initial values for these related values
must be on the stack, as with lua_pushc-
closure . Thus, to create a library where functions will be
are supposed to have a common table as the only associated value, we
we can use the following code:

```
/ * create a table for the library ('lib' is a list of its functions) * /
luaL_newlibtable (L, lib);
/ * create shared value * /
lua_newtable (L);
/ * add functions from the 'lib' list to the new library, so * /
/ * they will all have this table as their associated value * /
luaL_setfuncs (L, lib, 1)
```

The last call also pops the table off the stack, leaving
just a new library.

Exercises

Exercise 28.1. Write a C function filter . She gets-
takes a list and a function as input and returns all elements from
a given list for which the function returns true
value:

```
t = filter ({1, 3, 20, -4, 5}, function (x) return x <5 end)
- t = {1, 3, -4}
```

Exercise 28.2. Modify the l_split function (from Listing 28.2)
so that it can work with lines containing well-
left byte. (Apart from other changes, it must also
use memchr instead of strchr .)

Exercise 28.3. Implement the transliterate function (exercise
21.3) on C.

Exercise 28.4. Implement a library with modified functionality
it transliterate so that replacement of the table is not transmitted as
argument, but is stored by the library itself. Your library
should provide the following features:

```
lib.settrans (table) - set the replacement table
lib.gettrans () - return replacement table
lib.tranliterate (s) - translate 's' using the current table
```

Use a registry to store the replacement table.

Exercise 28.5. Repeat the previous exercise using
specifying the associated value to store the table.
Exercise 28.6. Do you think it is a good design to store
the replacement table as part of the library state, rather than passing
to use it as a parameter?
Exercises

Chapter 29
Asked user-defined types in C

In the previous chapter, we saw how to extend Lua with new
functions written in C. Now we will see how to extend
Lua with the new types defined in C. We'll start with a little
th example; throughout this chapter, we will expand it with
help of metamethods and other possibilities.
Our example will be pretty simple: an array of logical (boolean
out) values. Such a simple structure was chosen because with it
does not involve any complex algorithms and we can completely
concentrate on API. Nevertheless, this example is still
useful. Of course in Lua we can use tables
to implement arrays of boolean values. But in implementation on
With we will use one bit for each element, that is
we only need about 3% of the memory that would be needed-
for the corresponding table.
For our implementation, we need the following definitions:

```
#include <limits.h>
#define BITS_PER_WORD (CHAR_BIT * sizeof (unsigned int))
#define I_WORD (i) ((unsigned int) (i) / BITS_PER_WORD)
#define I_BIT (i) (1 << ((unsigned int) (i)% BITS_PER_WORD))
```

The BITS_PER_WORD constant is the number of bits in an unsigned string.
scrap number. Macro I_WORD evaluates a word that contains a bit
at the given index, and the macro I_BIT calculates the bit mask for
corresponding bit.
We will represent our arrays with the following
structures:

```
typedef struct NumArray {
int size;
unsigned int values [1]; / * mutable part * /
} NumArray;
```

We are declaring a values array of size 1, since C89 is not
allows you to declare arrays with size 0; in fact we will
allocate the required number of elements when allocating memory for
our array. The following expression calculates the total size for
our bitmap with n elements:
sizeof (NumArray) + I_WORD (n - 1) * sizeof (unsigned int)
(We subtract one from n , since in our structure we already
allocated space for one element.)

29.1. User data (userdata)

Our first task is to represent the structure
NumArray in Lua. Lua provides a special base type for
this: *userdata* . This type simply corresponds to an area, memory in
so we can store anything without any specific
operations.
The lua_newuserdata function allocates a block of memory for a given time
measure, pushes the corresponding Lua value onto the stack and returns
dedicated block address:
void * lua_newuserdata (lua_State * L, size_t size);
If for some reason you need to allocate a block of memory
otherwise, you can easily create the corresponding object
Lua with pointer size and remember there pointer to allocated
block. We will discuss this technique in Chapter 30.
Using the lua_newuserdata function, the function to create new
out arrays of boolean values looks like this:

```
static int newarray (lua_State * L) {
int i;
size_t nbytes;
NumArray * a;
int n = luaL_checkint (L, 1);
luaL_argcheck (L, n> = 1, 1, “invalid size”);
nbytes = sizeof (NumArray) + I_WORD (n - 1) * sizeof (unsigned int);
a = (NumArray *) lua_newuserdata (L, nbytes);
a-> size = n;
for (i = 0; i <= I_WORD (n - 1); i ++)
```

```
a-> values [i] = 0; / * initialize the array * /
return 1; / * the new object is already on the stack * /
}
```

Once the newarray function is registered in Lua, we can create new arrays using expressions like: a = array. new (1000) .

In order to write a value to our array, we will use Call expressions of the form: array.set (a, index, value) . Later we see how you can use metatables to support more the traditional syntax is a [index] = value . In both cases the function that writes the element to the array is the same. we we assume that, as is customary in Lua, indices start at 1:

```
static int setarray (lua_State * L) {
NumArray * a = (NumArray *) lua_touserdata (L, 1);
int index = luaL_checkint (L, 2) - 1;
luaL_argcheck (L, a! = NULL, 1, "'array' expected");
luaL_argcheck (L, 0 <= index && index <a-> size, 2,
"Index out of range");
luaL_checkany (L, 3);
if (lua_toboolean (L, 3))
a-> values [I_WORD (index)] | = I_BIT (index); / * set a bit * /
else
a-> values [I_WORD (index)] & = ~ I_BIT (index); / * remove a bit * /
return 0;
}
```

Since Lua can be used as a boolean to call any value, then we use luaL_checkany to to make sure there is some value for this parameter. If we call setarray with incorrect arguments, then we will we get the corresponding error messages:

```
array.set (0, 11, 0)
-> stdin: 1: bad argument # 1 to 'set' ('array' expected)
array.set (a, 1)
-> stdin: 1: bad argument # 3 to 'set' (value expected)
```

The following function returns the value at the given index:

```
static int getarray (lua_State * L) {
NumArray * a = (NumArray *) lua_touserdata (L, 1);
int index = luaL_checkint (L, 2) - 1;
luaL_argcheck (L, a! = NULL, 1, "'array' expected");
luaL_argcheck (L, 0 <= index && index <a-> size, 2,
"Index out of range");
lua_pushboolean (L, a-> values [I_WORD (index)] & I_BIT (index));
return 1;
}
```

We will define a separate function to return array size:

```
static int getsize (lua_State * L) {
NumArray * a = (NumArray *) lua_touserdata (L, 1);
luaL_argcheck (L, a! = NULL, 1, “'array' expected”);
lua_pushinteger (L, a-> size);
return 1;
}
```

Finally, we need additional code to initialize our library:

```
static const struct luaL_Reg arraylib [] = {
{“New”, newarray},
{“Set”, setarray},
{“Get”, getarray},
{“Size”, getsize},
{NULL, NULL}
```

```
};
int luaopen_array (lua_State * L) {
luaL_newlib (L, arraylib);
return 1;
}
```

Again we use the luaL_newlib function from the helper libraries. She creates a table and populates it with name-function pairs. set by the array arraylib .

After opening the library, we are ready to use our new type in Lua:

```
a = array.new (1000)
print (a)
-> userdata: 0x8064d48
print (array.size (a))
-> 1000
for i = 1, 1000 do
array.set (a, i, i% 5 == 0)
end
print (array.get (a, 10))
-> true
```

29.2. Metatables

Our current implementation has a big security problem. thu. Let's say the user writes something like a rray.set (io. stdin, 1, false) . The io.stdin value is an object of type userdata with a pointer to FILE . Due to the fact that this is also a userdata value , then array.set will take it as a valid argument; as a result we will most likely get a write to an arbitrary memory location (if we are very lucky, we will only receive a message about the recording by invalid index). This behavior is unacceptable for any fight the Lua library. No matter how you use the library edema, we shouldn't write something to an arbitrary memory address (or cause the entire application to crash).

The usual way to distinguish one type of userdata object from another is Go is to define a unique metatable for this type. Each time when we create an object of type userdata , we expose it to the corresponding metatable; every time we get an object of type userdata , we check that it has the correct metatable. Since Lua code cannot change the metatable for objects

type userdata , we are guaranteed to be fine.
We also need a place to store this metatable.
so that we can refer to it when creating new objects and
checks whether an object of type userdata has the type we need. Like us
said earlier, there are two options for storing the metatable: in the registry
or as a bound value for functions in a library. In Lua, when-
It is easy to register each new C type in the registry using
the type name as the index and the metatable as its corresponding
values. As with any other indexes in the registry, we
should choose the type name carefully to avoid possible
conflicts. In our example, we will use the name
"LuaBook.array" .
As usual, the helper library provides us with the functionality
tion. These new helper functions are as follows -
other functions:

```
int luaL_newmetatable (lua_State * L, const char * tname);
void luaL_getmetatable (lua_State * L, const char * tname);
void * luaL_checkudata (lua_State * L, int index,
const char * tname);
```

LuaL_newmettable function creates a new table (which will
det our metatable), puts it on the top of the stack and links
a table with a given name in the registry. LuaL_getmetatable function
returns the metatable associated with tname in the registry. At the same time
nets, the luaL_checkudata function checks that the object at the given
place on the stack is an object of type userdata with a metatable, with
corresponding to the given name. It raises an error if the object
that other metatable (or it doesn't exist) or is it not an object of type userdata ;
otherwise, it returns the address of the object.
Now we can start our implementation. The first step would be
changing the function that opens our library. New ver-
This should create a metatable for our arrays:

```
int luaopen_array (lua_State * L) {
luaL_newmetatable (L, "LuaBook.array");
luaL_newlib (L, arraylib);
return 1;
}
```

The next step is to change the newarray function to be
all at once so that it sets the metatable for the created mas-
Sivov:

```
static int newarray (lua_State * L) {
<as before>
```

```
luaL_getmetatable (L, “LuaBook.array”);
lua_setmetatable (L, -2);
return 1; / * the new object is already on the stack * /
}
```

The lua_setmetatable function pops a table off the stack and sets casts it as a metatable for an object on the stack by request given index. In our case, this object is the created an object of type userdata .

Finally, the setarray , getarray and getsize functions need to know did they actually get a valid array as their th first argument. To simplify this task, we will define next macro:

```
#define checkarray (L) \
(NumArray *) luaL_checkudata (L, 1, “LuaBook.array”)
```

Using this macro, the new getsize implementation becomes very simple:

```
static int getsize (lua_State * L) {
NumArray * a = checkarray (L);
lua_pushinteger (L, a-> size);
return 1;
}
```

Since setarray and getarray share common code to check index as our second argument, we will place the common parts in the next blowing function:

```
static unsigned int * getindex (lua_State * L,
unsigned int * mask) {
NumArray * a = checkarray (L);
int index = luaL_checkint (L, 2) - 1;
luaL_argcheck (L, 0 <= index && index <a-> size, 2,
“Index out of range”);
/ * return the address of the element * /
* mask = I_BIT (index);
return & a-> values [I_WORD (index)];
}
```

Following are the resulting implementations of setarray and getarray :

```
static int setarray (lua_State * L) {
unsigned int mask;
unsigned int * entry = getindex (L, & mask);
luaL_checkany (L, 3);
if (lua_toboolean (L, 3))
* entry | = mask;
else
* entry & = ~ mask;
return 0;
}
static int getarray (lua_State * L) {
```

```
unsigned int mask;
unsigned int * entry = getindex (L, & mask);
lua_pushboolean (L, * entry & mask);
return 1;
}
```

Now if you try to do something like array.
get (io.stdin, 10) , then you will receive a corresponding message about error:

```
error: bad argument # 1 to 'get' ('array' expected)
```

29.3. Object- oriented access

Our next step will be to convert our new type to an object
ect so that we can work with it using object-oriented
tied syntax as shown below:

```
a = array.new (1000)
print (a: size ())
-> 1000
a: set (10, true)
print (a: get (10))
-> true
```

Recall that a: size () is the same as a.size (a) . So-
we must make it so that a.size returns our function
getsize . The key mechanism here is the __index metamethod .
For tables, Lua calls this metamethod when it cannot find
values for the given key. For objects of type Lua userdata, you-
calls it every time it is accessed, since such objects have
there are no keys yet.
Let's assume we ran the following code:

```
local metaarray = getmetatable (array.new (1))
metaarray .__ index = metaarray
metaarray.set = array.set
metaarray.get = array.get
metaarray.size = array.size
```

In the first line, we create an array just to get
read its metatable, which we write to metarray . (We are not
we can set the metatable of an object of type userdata from Lua, but we
can get it.) Then we set the metaarray .__ index

equal to metaarray . Then when we execute a.size , Lua cannot
can find the key size in object a , since it is an object of type user-
data . So Lua tries to get this value from the __index field
metatable a , which is the same as the metaarray itself . But metaar-
ray.size is array.size , so a.size (a) returns array.
size (a) , which is what we wanted.
Of course we can do the same in C. We can do
even better: now that arrays are objects with their
our own operations, we no longer need to have these operations
tions in the array table . The only function from our library,
which we have to pass out is the new function to create
new arrays. All other operations will be available only
as methods. The C code can register them itself.
The getsize , getarray and setarray operations will not change compared to
with our previous approach. All that will change is how
we will register them. To do this, we need to change the code that
opens the library. First, we need two separate lists
functions: one for regular functions and one for methods.

```
static const struct luaL_Reg arraylib_f [] = {
{“New”, newarray},
{NULL, NULL}
};
static const struct luaL_Reg arraylib_m [] = {
{“Set”, setarray},
{“Get”, getarray},
{“Size”, getsize},
{NULL, NULL}
};
```

The new version of the opening function luaopen_array should match
create a metatable, assign it to its own __index field ,
register all methods and create and populate the array table :

```
int luaopen_array (lua_State * L) {
luaL_newmetatable (L, “LuaBook.array”);
/ * metatable .__ index = metatable * /
lua_pushvalue (L, -1); / * create a copy of the metatable * /
lua_setfield (L, -2, “__index”);
luaL_setfuncs (L, arraylib_m, 0);
luaL_newlib (L, arraylib_f);
return 1;
}
```

Here we again use luaL_setfuncs to write
put the functions from arraylib_m into the metatable located on

top of the stack. We then use luaL_newlib to
create a new table and register functions from arraylib_f
(actually just the new function).
As a finishing touch, we'll add the __tostring method to
our type so that print (a) prints "array" and the size of the array
in parentheses, something like “array (1000)” . The corresponding
the function is shown below:

```
int array2string (lua_State * L) {
NumArray * a = checkarray (L);
lua_pushfstring (L, “array (% d)”, a-> size);
return 1;
}
```

The lua_pushfstring call builds the string and leaves it at the top
stack. We also have to add array2string to the list of array-
lib_m , in order to include it in the corresponding metatable-
tsu:

```
static const struct luaL_Reg arraylib_m [] = {
{“__Tostring”, array2string},
<other methods>
};
```

29.4. Access as usual array

An alternative to the object-oriented way of writing is
The usual way of working with arrays. Instead of writing a: get (i), we
we can just write a [i] . In our example, this is pretty easy.
do as our setarray and getarray functions are already semi-
give their arguments in the order in which they should be passed
to use the appropriate metamethods. A quick fix would be
define these metamethods right in your Lua code:

```
local metaarray = getmetatable (array.new (1))
metaarray .__ index = array.get
metaarray .__ newindex = array.set
metaarray .__ len = array.size
```

(We have to execute this code for our original implementation
arrays, without modifications for the object-oriented syntax

sisa.) That's all we need to use the standard syntaxis:

```
a = array.new (1000)
a [10] = true - 'setarray'
print (a [10]) - 'getarray' -> true
print (#a) - 'getsize' -> 1000
```

If we want this, then we can register these metamethodes right in the C code. To do this, we must again change our initializing function:

```
static const struct luaL_Reg arraylib_f [] = {
{“New”, newarray},
{NULL, NULL}
};
static const struct luaL_Reg arraylib_m [] = {
{“__Newindex”, setarray},
{“__Index”, getarray},
{“__Len”, getsize},
{“__Tostring”, array2string},
{NULL, NULL}
};
int luaopen_array (lua_State * L) {
luaL_newmetatable (L, “LuaBook.array”);
luaL_setfuncs (L, arraylib_m, 0);
luaL_newlib (L, arraylib_f);
return 1;
}
```

In this version, we again have only one new function visible to everyone . All other functions are available only as metamethods for corresponding corresponding operations.

29.5. Light objects like userdata (light userdata)

The type of objects we have used so far is called *full userdata* . Lua offers another type of userdata object called light, - *light userdata* .

Such objects are just a pointer in C (i.e. value of type void *). It is a value, not an object; we do not create them (just like we don't create numbers). In order to place such object on the stack, we call lua_pushlightuserdata :

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

Despite the common name, full and lightweight objects like userdata is actually quite different. Light objects are not buffers, just pointers. They don't have metatables. Like numbers, they not managed by the garbage collector.

Sometimes we use the lightweight option as a cheap alternative. full-fledged objects of type userdata . However, this is not their typical using. First, light objects do not have metatables, so we cannot find out their type. Secondly, despite its name full-fledged userdata objects are pretty cheap. They add They have very little overhead compared to calling malloc .

The real use of light objects comes from equality. A fully-fledged userdata object is equal only to itself. Lay down cue object is just a pointer. And as such he is equal to any other object of type userdata representing that the same pointer. This way we can use the lungs objects of type userdata so that C objects are inside Lua.

We have already seen the typical use of lightweight objects as a key whose in the registry (see section 28.3). There the equality of light objects was extremely important. Every time we push a light object onto the stack with lua_pushlightuserdata , we get the same value Lua and, accordingly, the same entry in the registry.

Another typical scenario is the need to obtain
a full-fledged userdata object at its address in C. Let's say we
we organize the connection between Lua and the window system. Then we can use
use full-fledged userdata objects for presentation
windows. Each such object contains or the entire structure, representing
a window, or just a pointer to a structure created by the system.
When an event occurs inside the window (for example, clicking a button
mouse), the system calls the appropriate handler that identifies
quoting the window at its address. In order to pass a Lua handler,
we have to find an object of type userdata representing the given
window. In order to find it, we can use the table,
where indices are light objects containing window addresses,
and the values are full-fledged objects of the userdata type ,
setting the appropriate windows. If we have a window address, we
push it onto the stack as a light object of type userdata and use
its as an index on the table. (Most likely this table should have
weak values. Otherwise, they will never be collected by the collector.
garbage.)

Exercises

Exercise 29.1. Modify the setarray implementation so that it
took only boolean values as input.

Exercise 29.2. We can consider a boolean array
as a set of integers (indices that correspond to
are true values in the array). Add bu-
left array functions that compute union and
intersection of two arrays. These functions should receive
to input two arrays and return a new array without changing
input arrays.

Exercise 29.3. Change the implementation of the __tostring metamethod
so that it shows the full contents of the array by some
either way. Use buffers (see Section 28.2) to co-
building the summary line.

Exercise 29.4. Based on the example with boolean arrays, re-
Alize the C library for working with arrays of integers
sat down.

Chapter 30
Resource management

In our implementation of boolean arrays from the previous chapter, we do not worried about resource management. These arrays use only memory. Each userdata object that represents a mass siv, has its own block of memory, which is managed by Lua. When the array becomes garbage (that is, no one stores references to it), Lua with will collect it in time and free the occupied memory.

However, life is not always so easy. Sometimes the object needs others resources other than memory such as file descriptors, pointers to windows, etc. (often these resources are also memory, but it is controlled other part of the system). In such cases, when the object becomes curled with garbage, it is necessary to somehow release these resources.

As we saw in section 17.6, Lua provides a finalizer __gc metametode . To show the use of this metamethod in C, we implement two libraries in C, providing access to external resources. The first example is another re-Alization of the function for crawling the contents of the directory. The second (and more complex) example is using the *Expat* library to parse XML files.

30.1. Directory iterator

In section 27.1, we implemented a dir function that returned a table with all files from a given directory. Our new reality zation will return an iterator that returns a new file each time call. Using this implementation, we can iterate over the content directory using a loop as shown below:

```
for fname in dir.open (“.”) do
print (fname)
end
```

In order to iterate over the contents of the directory in C, we need

structure DIR . These structures are created by calling opendir
and are destroyed by calling closedir . Our previous
the implementation of the dir function kept this structure as local to
belt and released when getting the name of the last file.
Our new implementation cannot store DIRs in local variables-
noah, since this structure will be needed for a whole series of
calls. Moreover, we cannot destroy it upon receiving the name
nor the last file, as the program may prematurely
get out of the loop, in which case we'll never get to the last
file. Therefore, in order to ensure that this structure will
det is always freed, we need to store its address in an object like
userdata and use the __gc metamethod to free this
structures.
Despite its central role in our implementation, this
an object representing a directory does not have to be
den from Lua. The dir function returns an iterating function; this is
whatever Lua sees. The directory can be the associated value of this
iterating function. In this case, the iterating function will be
have direct access to this structure, but Lua code to it
does not have access (and he does not need it).
In total, we need three functions in C. First, we need dir.
open is a function that Lua calls to create iterators; it
should create a DIR structure and an iteration function closure with
by this structure (as a bound value). Second, we need
iterating function. Third, we need the __gc metamethod , which
ry releases the created DIR structure . As usual, we also
you will need a function for initial setup, such as creating
and initializing the metatable for the directory.
Let's start our code with the dir.open function shown in fox
tinge 30.1. The important point is that this function should
create a userdata object before opening the directory. Otherwise, if he
will open the directory first and then calling lua_newuserdata will result in
error when working with memory, then a memory leak occurs, since
nobody will release the created structure. With the right order
the DIR structure , as soon as it is created, is immediately linked to the
ektu the userdata ; whatever happens after, the __gc metamethod over time
it will release this structure.
The next function is dir.iter (Listing 30.2), the iterator itself.
Its code is pretty simple. It gets the address of the DIR structure from the

associated
value with it and calls readdir to get the next
values.
The dir_gc function (also in Listing 30.2) is the __gc metamethod . is he
frees the generated DIR structure, but you need to be careful:
since the userdata object is created before opening the directory, but
even if opendir returns an error, then the userdata object is all
will be created equal. Therefore, we need to check what is, what
close.
The last function in Listing 30.2 is luaopen_dir , a function that
which opens our library.
There is one subtlety in the complete example. At first it may seem
that the function dir_gc needs to know if its argument is valid
is a directory. Otherwise, the user can call
its with a different type of userdata object (like file), which will result
to a serious error. However, a Lua program has no way to
Refer to this function: it is stored as a metatable of catalogs,
which, in turn, are stored as associated with the iterating
value function. Lua programs can't access
to objects of this type.

Listing 30.1. Dir.open function

```
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include “lua.h”
#include "lauxlib.h"
/ * declare an iteration function * /
static int dir_iter (lua_State * L);
static int l_dir (lua_State * L) {
const char * path = luaL_checkstring (L, 1);
/ * create a userdata object to store the address of the DIR * /
DIR ** d = (DIR **) lua_newuserdata (L, sizeof (DIR *));
/ * set metatable * /
luaL_getmetatable (L, “LuaBook.dir”);
lua_setmetatable (L, -2);
/ * trying to open the directory * /
* d = opendir (path);
if (* d == NULL) / * error opening directory? * /
luaL_error (L, “cannot open% s:% s”, path, strerror (errno));
/ * create and return an iteration function;
its associated value is the userdata object,
already on the stack * /
```

```
lua_pushcclosure (L, dir_iter, 1);
return 1;
}
```

Listing 30.2. Other functions in the dir library

```
static int dir_iter (lua_State * L) {
DIR * d = * (DIR **) lua_touserdata (L, lua_upvalueindex (1));
struct dirent * entry;
if ((entry = readdir (d))! = NULL) {
lua_pushstring (L, entry-> d_name);
return 1;
}
else return 0; / * no more values * /
}
static int dir_gc (lua_State * L) {
DIR * d = * (DIR **) lua_touserdata (L, 1);
if (d) closedir (d);
return 0;
}
static const struct luaL_Reg dirlib [] = {
{“Open”, l_dir},
{NULL, NULL}
};
int luaopen_dir (lua_State * L) {
luaL_newmetatable (L, “LuaBook.dir”);
/ * set field __gc * /
lua_pushcfunction (L, dir_gc);
lua_setfield (L, -2, “__gc”);
/ * create library * /
luaL_newlib (L, dirlib);
return 1;
}
```

30.2. XML parser

We now turn to a simplified library implementation for links between Lua and the Expat library, which we will call lxp . Expat is an open source XML 1.0 parser written in C. It implements SAX, that is, a *simple API for XML* (simple API for XML). SAX is event-driven API. This means that the SAX parser is clean the XML document melts and tells the application as it reads that it finds using user - defined *functions-process-*

chikov (callback). For example, if we want Expat to parse
line like “<tag cap =” 5 ”> hi </tag>” , then it will create three events:
start event when it reads the line “<tag cap =” 5 ”>” ; event
text when it reads "hi" , and the end of element event when it reads
“</tag>” melts . Each of these events triggers a corresponding
handler in the application.
We won't cover the entire Expat library here. We focus
we focus only on those parts that show newer methods of interaction
Modeling with Lua. While Expat handles over a dozen different
events, we will consider only those three events that we saw in
previous example (start of element, end of element and text) 1 .
The part of the Expat API that we need is pretty small. In-
First, we need functions to create and destroy the parser:

```
XML_Parser XML_ParserCreate (const char * encoding);
void XML_ParserFree (XML_Parser p);
```

The encoding argument is optional, we will pass it instead
to be NULL.
Once we have a parser, we must register
our handler functions:

```
void XML_SetElementHandler (XML_Parser p,
XML_StartElementHandler start,
XML_EndElementHandler end);
void XML_SetCharacterDataHandler (XML_Parser p,
XML_CharacterDataHandler hndl);
```

The first function sets handlers for start and end events
element. The second function sets the handler for the text.
All handlers receive a non-
which is a pointer. The element start handler is also named
tag and its attributes:

```
typedef void (* XML_StartElementHandler) (void * uData,
const char * name,
const char ** atts);
```

Attributes are passed as a NULL terminated array of strings , where
each pair of consecutive lines contains an attribute and its value
nie. The end-of-element handler only receives one extra
nth element - tag name:

```
typedef void (* XML_EndElementHandler) (void * uData,
const char * name);
```

Finally, the text processor receives as an additional
parameter is the text itself. The line of text is not null terminated, and
the length is explicitly passed for it:

```
typedef void (* XML_CharacterDataHandler) (void * uData,
const char * s,
int len);
```
In order to pass text to Expat for parsing, we use
the following function:
```
int XML_Parse (XML_Parser p, const char * s, int len, int isLast);
```
Expat receives the document to be parsed hourly
cham through successive calls to XML_Parse . The last argument
such a call to isLast tells Expat whether the passed chunk was
last in the document. Note that each snippet
the text does not have to be terminated with a null byte, we explicitly
we pass its length. XML_Parse function returns zero in case
errors. (Expat also provides functions for getting information
error messages, but for simplicity, we will not consider them here.
vat.)
The last function we need from Expat is the function
ttion that allows you to specify the pointer that will be passed
handlers:
```
void XML_SetUserData (XML_Parser p, void * uData);
```
Now let's see how we can use this bib-
library in Lua. The first approach is the simplest: let's just yes-
Let's access all these functions from Lua. More successful will be
adapt this functionality for Lua. For example, since
Lua is an atypical language (more precisely, a language without strong typing),
then we
no need for different functions for each type of handler. More
Moreover, we can avoid registering handlers altogether. Inmes-
then we will create a parser, pass it a table of handlers,
each with a matching key. For example, if we want to print
structure of the document, then we can use the following table
zu handlers:
```
local count = 0
callbacks = {
StartElement = function (parser, tagname)
io.write (“+“, string.rep (““, count), tagname, “\ n”)
count = count + 1
end,
EndElement = function (parser, tagname)
count = count - 1
io.write (“-“, string.rep (““, count), tagname, “\ n”)
end,
}
```

If we give the input the string “<to> <yes /> </to>” , then these
bots will generate the following output:

```
+ to
+ yes
- yes
- to
```

With such an API, we do not need functions to work with the handler-
mi. We work with them directly in the handler table.
Thus, the entire API will consist of only three functions: one
for creating parsers, one for processing a piece of text and one
to destroy the parser. In fact, we are implementing two last
function as parser methods. As a result, we come to the following -
typical usage of our API:

```
local lxp = require ”lxp”
p = lxp.new (callbacks)
- create a new parser
for l in io.lines () do
- process input lines
assert (p: parse (l))
- parse the string
assert (p: parse (“\ n”))
- add '\ n'
end
assert (p: parse ())
- complete the document
p: close ()
```

Let's now turn to the implementation. Our first decision
This will be how we will represent our parser in Lua. Quite
it is natural to use an object of type userdata for this , but what
do we need to put inside it? At least we need
the parser itself and the handler table. We cannot remember the table
zu inside an object of type userdata (or inside a C structure), but Lua
allows each object of type userdata to have a *custom*
a user value, which can be any Lua 2 table . we
should also remember the Lua state into the parser object as
all that the Expat handler receives is the parser itself, and in order to
call Lua, we need this state. Therefore, we will use
define the following parser:

```
#include <stdlib.h>
#include “expat.h”
#include “lua.h”
#include "lauxlib.h"
typedef struct lxp_userdata {
XML_Parser parser; / * corresponding Expat parser * /
```

```
lua_State * L;
} lxp_userdata;
```
2
In Lua 5.1, the environment of the userdata object acts as a user value.
niya.

Our next step is to create a function that creates
no parsers, lxp_make_parser . Its code is shown in Listing 30.3. This
the function consists of four important steps:

• Its first step follows a standard pattern: first create
a userdata object ; then it is initialized appropriately
values, and finally, a metatabe is assigned to it.
faces. The reason for this initialization is as follows: if in
any error occurs during initialization, you must
dimo that the finalizer (metamethod __gc) finds our data
holistic.

• At step 2, the function creates an Expat parser, stores it in
userdata object and checks for errors.

• Step 3 checks that the first argument of the function is valid
a table (table of handlers) is stored, and
assigns it as a custom value for the userdata object.

• The last step initializes the Expat parser. Our userdata is
the object is given as a pointer to be passed
to be included in handlers, handler functions are also set.
Note that these handlers are the same for everyone
parsers; after all, in C one cannot dynamically construct
function. Instead, fixed functions using
a table of handlers, decide which Lua functions to follow
call.

Listing 30.3. Function for creating XML parsers

```
/ * descriptions of handler functions * /
static void f_StartElement (void * ud,
const char * name,
const char ** atts);
static void f_CharData (void * ud, const char * s,
int len);
static void f_EndElement (void * ud, const char * name);
static int lxp_make_parser (lua_State * L) {
XML_Parser p;
/ * (1) create parser object * /
lxp_userdata * xpu = (lxp_userdata *)
lua_newuserdata (L,
sizeof (lxp_userdata));
/ * initialize it in case of error * /
```

```
xpu-> parser = NULL;
/ * set a metatable for it * /
luaL_getmetatable (L, “Expat”);
lua_setmetatable (L, -2);
/ * (2) create Expat parser * /
p = xpu-> parser = XML_ParserCreate (NULL);
if (! p)
luaL_error (L, “XML_ParserCreate failed”);
/ * (3) check and save the handler table * /
luaL_checktype (L, 1, LUA_TTABLE);
lua_pushvalue (L, 1); / * push the table onto the stack * /
lua_setuservalue (L, -2);
/ * (4) configure the Expat parser * /
XML_SetUserData (p, xpu);
XML_SetElementHandler (p, f_StartElement,
f_EndElement);
XML_SetCharacterDataHandler (p, f_CharData);
return 1;
}
```

Listing 30.4. Function for parsing a piece of text

```
static int lxp_parse (lua_State * L) {
int status;
size_t len;
const char * s;
lxp_userdata * xpu;
/ * get and check the first argument * /
xpu = (lxp_userdata *) luaL_checkudata (L, 1, “Expat”);
/ * check that it is not closed * /
luaL_argcheck (L, xpu-> parser! = NULL, 1, “parser is closed”);
/ * get second argument (string) * /
s = luaL_optlstring (L, 2, NULL, & len);
/ * put the handler table at index 3 on the stack * /
lua_settop (L, 2);
lua_getuservalue (L, 1);
xpu-> L = L; / * set Lua state * /
/ * call Expat to parse the string * /
status = XML_Parse (xpu-> parser, s, (int) len, s == NULL);
/ * return error code * /
lua_pushboolean (L, status);
return 1;
}
```

The next step is the method for parsing the text lxp_parse (Listing 30.4), which parses a chunk of XML data. He semi-takes two arguments: a parser (*self* in the method) and an optional fragment XML. When called with no data, it tells Expat that more

there are no parts.
When lxp_parse calls XML_Parse it will call handlers
for those elements that it finds in the transferred text fragment
that. These handlers will need access to the handler table,
so lxp_parse pushes this table onto the stack at index 3 (immediately
after parameters). There is one caveat to the XML_Parse call : remember,
that the last argument to this function tells Expat whether
the last piece of text transferred. When we call parse
with no arguments, s will be NULL , and this last argument will be
met the true value.
Now let's turn our attention to the handler functions
f_StartElement , f_EndElement and f_CharData . All these functions are
have the same structure: each of them checks if there is
handler table for this event, and if such
a handler is present, then prepares the arguments and then you-
calls this handler.
Let's take a look at the f_CharData handler in listing
ge 30.5. Its code is pretty simple. The handler receives the structure
lxp_userdata as its first argument, since we called
XML_SetUserData when we created our parser. After receiving co-
Lua state handler can refer to handler table
on the stack at index 3, given by lxp_parse , and the parser itself by
index 1. Then it calls the corresponding Lua handler
(when present) with two arguments: parser and character data
nym (string).

Listing 30.5. Character data processor

```
static void f_CharData (void * ud, const char * s, int len) {
lxp_userdata * xpu = (lxp_userdata *) ud;
lua_State * L = xpu-> L;
/ * get handler * /
lua_getfield (L, 3, “CharacterData”);
if (lua_isnil (L, -1)) {/ * no handler? * /
lua_pop (L, 1);
return;
}
lua_pushvalue (L, 1); / * push the parser ('self') onto the stack * /
lua_pushlstring (L, s, len); / * push a line onto the stack * /
lua_call (L, 2, 0); / * call handler * /
}
```

The f_EndElement handler is quite similar to f_CharData ; education
See Listing 30.6. It also calls the appropriate
Lua bot with two arguments - a parser and a tag name (again

a string, this time terminated with a null byte).

Listing 30.6. End of element handler

```
static void f_EndElement (void * ud, const char * name) {
lxp_userdata * xpu = (lxp_userdata *) ud;
lua_State * L = xpu-> L;
lua_getfield (L, 3, “EndElement”);
if (lua_isnil (L, -1)) {/ * no handler? * /
lua_pop (L, 1);
return;
}
lua_pushvalue (L, 1); / * push the parser ('self') onto the stack * /
lua_pushstring (L, name); / * push the tag onto the stack * /
lua_call (L, 2, 0); / * call handler * /
}
```

Listing 30.7 shows the final handler, f_StartElement .
It calls Lua with three arguments: parser, tag name, and
list of attributes. This handler is slightly more complex than the others, because how long it is necessary to translate the list of attributes into Lua. He used suggests a completely natural translation, building a table that compares Specifies attribute names and their values. For example, for a short tag, shown below

<to method = ”post” priority = ”high”>

the following attribute table is generated:

{method = “post”, priority = “high”}

Listing 30.7. Element start handler

```
static void f_StartElement (void * ud,
const char * name,
const char ** atts) {
lxp_userdata * xpu = (lxp_userdata *) ud;
lua_State * L = xpu-> L;
lua_getfield (L, 3, “StartElement”);
if (lua_isnil (L, -1)) {/ * no handler? * /
lua_pop (L, 1);
return;
}
lua_pushvalue (L, 1); / * push the parser ('self') onto the stack * /
lua_pushstring (L, name); / * push the tag name onto the stack * /
/ * create and populate the attribute table * /
lua_newtable (L);
for (; * atts; atts + = 2) {
lua_pushstring (L, * (atts + 1));
```

```
lua_setfield (L, -2, * atts); / * table [* atts] = * (atts + 1) * /
}
lua_call (L, 3, 0); / * call handler * /
}
```

The final method for parsers is close , shown in listing
ge 30.8. When we close the parser, we must release all of it
resources, namely the Expat structure. Remember that due to errors in
the parser may not have this structure. Pay attention-
how we keep the parser consistent across
as we close it, so there will be no problem,
if we try to close it again or when the garbage collector
finalizes it. This ensures that each parser over time
will free its resources even if the programmer hasn't closed it.

Listing 30.8. Method for closing an XML parser

```
static int lxp_close (lua_State * L) {
lxp_userdata * xpu =
(lxp_userdata *) luaL_checkudata (L, 1, “Expat”);
/ * free the Expat parser (if any) * /
if (xpu-> parser)
XML_ParserFree (xpu-> parser);
xpu-> parser = NULL; / * if we close it again * /
return 0;
}
```

The final step is shown in Listing 30.9: it shows the functionality
tion luaopen_lxp , which opens up the library, combining together
all previously discussed functions. We use the same scheme here
mu used for object-oriented boolean
array in section 29.3: we create a metatable, set it
the __index field on it and put all methods inside it. For this-
Then we need a list with all the parser methods (lxp_meths).
We also need a list of functions of this library (lxp_funcs). how
and is accepted in object-oriented libraries, this list is co-
holds just one function that creates new parsers.

Listing 30.9. Initializing code for lxp library

```
static const struct luaL_Reg lxp_meths [] = {
{“Parse”, lxp_parse},
{“Close”, lxp_close},
{“__Gc”, lxp_close},
```

```
{NULL, NULL}
};
static const struct luaL_Reg lxp_funcs [] = {
{“New”, lxp_make_parser},
{NULL, NULL}
};
int luaopen_lxp (lua_State * L) {
/ * create metatable * /
luaL_newmetatable (L, “Expat”);
/ * metatable .__ index = metatable * /
lua_pushvalue (L, -1);
lua_setfield (L, -2, “__index”);
/ * register methods * /
luaL_setfuncs (L, lxp_meths, 0);
/ * register functions (lxp.new only) * /
luaL_newlib (L, lxp_funcs);
return 1;
}
```

Exercises

Exercise 30.1. Modify the dir_iter function so that-
would it close the DIR structure when it reaches the end
directory. With this change, the program does not need to wait
garbage collection to release a larger resource
not needed.
(When you close the directory, you must set the address,
written in the userdata object to NULL to inform the ph-
the analyzer that the directory is already closed. Also the dir_iter function
before using the directory should check that it is not
closed.)

Exercise 30.2. In the example with the lxp library , the handler starts
la element gets a table with the attributes of the element. In this
the table is the order in which the elements were given internally
item is already lost. How can you convey this information
to the handler?

Exercise 30.3. In the example with the lxp library , we used
whether a custom value for linking the table
handlers with the corresponding userdata object , pre-
installing the parser. This choice created a small problem,

since what the C handlers get is the structure
lxp_userdata , and this structure does not provide direct access to
stupa to this table. We solved this problem by co-
storing the handler table at a given place on the stack in
parsing time of each fragment.
Another solution could be to link the handler table
with a userdata object using links (section 28.3): we
create a link to the handler table and remember this
link (integer) in the lxp_userdata structure . Implement
this option. Don't forget to release the link on close
parser.

CHAPTER 31
Threads and states

Lua does not support true multi-threading, that is, it displaces
threads sharing shared memory. There are two reasons for this.
The first reason is that such support was not provided.
is ANSI C and therefore there is no portable way to implement this
support in Lua. The second and more serious reason is
that we don't think multi-threading is a good idea for Lua.
Multi-threading was developed for low-level production
gramming. Synchronization mechanisms like semaphores and
nitors have been proposed for operating systems (and experienced
programmers), not for applications. It is extremely difficult to find and
fix bugs related to multi-threading and some of
they can lead to security holes. Also multi-threading
can lead to serious performance problems due to the need
synchronization at a number of critical points in the program, such
like memory allocation.
Multi-threading problems arise from the combination of displacement
threads and shared memory, so we can avoid

them, either without using displacing threads, or without using storage space. Lua offers support for both.
Lua threads (also known as coroutines) are not preemptive and therefore avoid the problems associated with unpredictable by switching the threads. Lua states have no shared memory, so form a good basis for parallel computing. In this chapter we will consider both of these options.

31.1. Numerous threads

A thread is the essence of a coroutine in Lua. We are considering a coroutine as a thread and a user-friendly interface, or we can consider a thread as a coroutine with a low-level API.
From a C perspective, it can be helpful to think of a thread as a ke - what the thread really is from the point of view of implementation. Each stack stores information about the current calls to the thread, as well as the same parameters and local variables of each call. Others in words, the stack contains all the information a thread needs to continuation of its implementation. Therefore, many threads mean many independent stacks.
When we call most of the functions from the Lua-C API, then these functions work with a specific stack. For example lua_ pushnumber must push a number onto a specific stack; also lua_pcall needs a stack to call. How does Lua know which stack to follow use? The secret is that the lua_State type , the first the argument of all these functions, is not only a state Lua, but also a thread inside that state. (Many believe that this type should be called lua_Thread .)
When you create a new state, Lua automatically creates thread inside this state and returns lua_State representing that thread. This *main thread is* destroyed along with the state, when you call lua_close .
You can create other threads within the state with lua_newthread :

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

This function returns a pointer to lua_State representing the new thread, and also pushes the new thread onto the stack as a value of type thread . For example, after executing the operator

```
L1 = lua_newthread (L);
```
we will have two strands, L1 and L, both referring internally to the same
the same state of Lua. Each thread has its own stack. New
thread L1 starts from an empty stack; old thread L has new thread on
top of stack:
```
printf (“% d \ n”, lua_gettop (L1)); -> 0
printf (“% s \ n”, luaL_typename (L, -1)); -> thread
```
With the exception of the main yarn, the yarns can be collected by the picker.
a lump of garbage like any other Lua object. When you create a new
thread, then it is pushed onto the stack, which ensures that this thread is not
rubbish. You should never use a thread that
not tied to state. (The main thread is tied from the beginning,
so you don't have to worry about it.) Any Lua API call can
destroy an unattached thread, even a call using this self-
wash the thread. For example, let's look at the following snippet:
```
lua_State * L1 = lua_newthread (L);
lua_pop (L, 1); / * L1 is now garbage for Lua * /
lua_pushstring (L1, “hello”);
```
Calling lua_pushstring can invoke the garbage collector and collect
L1 (resulting in an application error) even though L1 is still in use
enjoys. To avoid this, always keep links to
the threads you are using, for example on the anchored thread stack
or in the registry.
As soon as we have a new thread, we can immediately start
use it like the main thread. We can put values
onto its stack and pop values from its stack, we can use it to
function calls, etc. For example, the following code calls
f (5) on the new thread and then puts the result on the old thread:
```
lua_getglobal (L1, “f”); / * consider that there is a global 'f' * /
lua_pushinteger (L1, 5);
lua_call (L1, 1, 1);
lua_xmove (L1, L, 1);
```
The lua_xmove function moves a Lua value between two stacks-
the same state. Calling type lua_xmove (F, T, n) from below
meth n elements from the stack F and places them on a stack T .
However, we do not need a new thread for these purposes; we can light
to use the main thread. The main purpose of using non-
how many threads is the implementation of coroutines so that we can
suspend their execution and resume it again. For this
we need the lua_resume function :
```
int lua_resume (lua_State * L, lua_State * from, int narg);
```

To start the execution of the coroutine, we use lua_resume
just like we use lua_pcall : we put the function on
stack, push its arguments onto the stack and call lua_resume , re-
giving nargs the number of arguments. (The from parameter is the thread that
makes the call.) This is very similar to lua_pcall , however there are three
differences. First, lua_resume does not contain a parameter for the number
desired results; it always returns all the results you-
called function. Secondly, it has no parameter for the handler
errors; the error does not unwind the stack, so you can later
explore. Third, if a function suspends its execution,
(with yield), then lua_resume returns a special
code LUA_YIELD and leaves the thread in such a state that we can
resume later.
When lua_resume returns LUA_YIELD , the visible portion of the
ka thread contains only the values passed to yield . Lua_ call
gettop will return a number of these values. In order to transfer these
values to another thread, we can use lua_xmove .
To continue execution of a suspended thread,
we call lua_resume again . In this case, Lua assumes that everything knows
the values on the stack are the values returned by
yield . For example, if you don't touch the stack thread in between returns
eat from the previous lua_resume and the next lua_resume , then yield
will return exactly the values with which it was called.
We usually run a coroutine with a Lua function as
body. This Lua function can call other functions, and any
from these functions can call yield , ending the call to lua_resume .
For example, consider the following definitions:

```
function foo (x) coroutine.yield (10, x) end
function foo1 (x) foo (x + 1); return 3 end
```

We will now execute the following C code:

```
lua_State * L1 = lua_newthread (L);
lua_getglobal (L1, “foo1”);
lua_pushinteger (L1, 20);
lua_resume (L1, L, 1);
```

Call lua_resume return LUA_YIELD , to inform,
that the thread is suspended. At this point, the L1 stack contains the values
passed to yield :

```
printf (“% d \ n”, lua_gettop (L1)); -> 2
printf (“% d \ n”, lua_tointeger (L1, 1)); -> 10
printf (“% d \ n”, lua_tointeger (L1, 2)); -> 21
```

When we call lua_resume again , the thread will continue executing

from where it left off (call to yield). From there foo will return the control Lenie foo1 , and she, in turn, returns management lua_resume :

```
lua_resume (L1, L, 0);
printf ("% d \ n", lua_gettop (L1)); -> 1
printf ("% d \ n", lua_tointeger (L1, 1)); -> 3
```

This second call to lua_resume will return LUA_OK which means ok minimal return.
Coroutines can also call C functions that can call other Lua functions. We have already discussed how to use continue to allow these functions in Lua

call yield (section 27.2). A C function itself can also call yield . In this case, you must provide a continuation function, which will be called when execution continues. On the C following- This function plays the role of yield :

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

We must always use this function in a return statement the one as shown below:

```
static inf myCfunction (lua_State * L) {
...
return lua_yieldk (L, nresults, ctx, k);
}
```

This call immediately suspends the currently running program. The nresults parameter is the number of values on the stack, which should be returned to the corresponding lua_resume ; ctx is the context to be passed on to the continuation and k is a function-continuation. When the coroutine continues execution, the control the extension goes to the continuation function k . After calling lua_yieldk the myCfunction cannot do anything else; she must delegate all further work to your continuation.
Let's look at a hypothetical example. Let us want write a function that reads some data by calling yield when the data is not ready. We can write this function in With the following:

```
int prim_read (lua_State * L) {
if (nothing_to_read ())
```

```
return lua_yieldk (L, 0, 0, & prim_read);
lua_pushstring (L, read_some_data ());
return 1;
}
```

If the function has any data, then it reads and returns
them. Otherwise, it calls lua_yieldk . When the thread is
continues to execute, it will call the continuation function. In that
in the example, the continuation function is prim_read itself , so the thread
will call it over and over again to read data. (This template,
when the calling lua_yieldk function is itself
position is not uncommon.)
If a C function has nothing to do after calling lua_yieldk , then it
can call lua_yieldk without a continuation function or using
lua_yield macro :

```
return lua_yield (L, nres);
```

After this call, when the thread continues its execution,
raids the function that called myCfunction .

31.2. Lua states

Each call to luaL_newstate (or lua_newstate , as we will see
in chapter 32) creates a new Lua state. Different states of Lua
do not depend on each other in any way. And they don't have any common data.
This means that no matter what happens in one state
in Lua, it cannot "harm" another state in any way. So-
this means that different states of Lua cannot be
communicate; for this we have to use a special code on
C. For example, if we have two states L1 and L2 , then the following
command - push to the top of the L2 stack the value from the top of the stack
in L1:

```
lua_pushstring (L2, lua_tostring (L1, -1));
```

Since the data must pass through C, different co-
states in Lua can exchange only those types
data that is representable in C, such as strings and numbers. Other types,

for example, tables to be transferred must be serialized.
In systems that offer multi-threading, an interesting archi-
the architectural solution would be to create on a separate state Lua
for each thread. As a result, we get threads that behave
like processes in UNIX, that is, we have parallelism without
shared (shared) memory. In this section, we will build a prototype
applications using this approach. For this implementation I
I will use POSIX threads (pthreads). Since I am using
only the most basic features, it will be easy to transfer
this code to other multi-threaded systems.
The system we want to build is very simple. Its purpose is
to show the use of multiple Lua states in a context
ste multi-threading. After it is ready, you yourself can
add additional features to it. We will call our
lproc library . It offers just four functions:
• lproc.start (chunk) starts a new process to execute
the given block of code (chunk). The library implements the
cess in Lua as a C thread and associated Lua state.
• lproc.send (channel, val1, val2, ...) sends the given values
reads (which must be strings) to a given channel,
identified by its name (string).
• lproc.receive (channel) receives values from the given channel
nala.
• lproc.exit () ends the process. This function is only needed
the main process. If this process ends without calling
lproc.exit , then the entire program terminates
without waiting for other processes to finish.
The library identifies channels using strings and using
calls them to match the sender of the recipient. Opera-
the radio can send any number of string values
nii that are returned by the corresponding receive operation
data. All interaction is synchronous: the process sending the message
feed into pipe, waits until there is a process reading from this
pipe while the process reading from the pipe is also waiting
until there is a process sending to it.
The lproc library , like its interface, is pretty simple. She is using
uses two doubly linked ring lists, one for processes,
waiting to send a message, and another waiting to send
processes waiting to receive a message. Also used one

a mutex to control access to both of these lists. Everyone has it process has its own *condition variable* . When the process wants to send a message to the channel, it looks in the waiting list the process that is waiting for this particular channel. If he finds such a process, then it removes it from the waiting list, transfers values from itself to the found process and signals the mental processes. Otherwise it inserts itself into the list sending and waiting for its conditional variable. Getting co-communication behaves similarly.
The main element of the implementation is a structure that represents the following process:

```
#include <pthread.h>
#include “lua.h”
typedef struct Proc {
lua_State * L;
pthread_t thread;
pthread_cond_t cond;
const char * channel;
struct Proc * previous, * next;
} Proc;
```

The first two fields represent the Lua state used by process, and the corresponding C thread performing this process cess. Other fields are used only when the process has to wait corresponding send / receive . The third field cond is a conditional the variable that the thread uses to wait; four-
The th field is the channel the process is waiting for; and the last two fields, previous and next are used to connect a process in a double-link list.
The following code declares two lists of waiting processes and their associated mutex:

```
static Proc * waitsend = NULL;
static Proc * waitreceive = NULL;
static pthread_mutex_t kernel_access = PTHREAD_MUTEX_INITIALIZER;
```

Each process needs a corresponding Proc structure , and it needs access to it whenever his body calls send or receive. The only parameter these functions receive is is the corresponding Lua state, so each process has to remember its Proc structure inside its Lua state.
In our implementation, each Lua state stores a corresponding

the Proc structure as an object of type userdata associated with the key "_SELF" . The getself helper function returns state Proc corresponding to the given state:

```
static Proc * getself (lua_State * L) {
Proc * p;
lua_getfield (L, LUA_REGISTRYINDEX, “_SELF”);
p = (Proc *) lua_touserdata (L, -1);
lua_pop (L, 1);
return p;
}
```

The next function, movevalues , transfers values from send-process to receiving:

```
static void movevalues (lua_State * send, lua_State * rec) {
int n = lua_gettop (send);
int i;
for (i = 2; i <= n; i ++) / * transfer values to the receiver * /
lua_pushstring (rec, lua_tostring (send, i));
}
```

It transfers to the receiver all values from the sender's stack, except for the first value, which is a channel.

Listing 31.1 defines a searchmatch function that bypasses pending list looking for a process waiting on the given channel

la. If the function finds such a channel, then it removes it from the list and returns it, otherwise it returns NULL .

Listing 31.1. Function to find a process waiting for a given channel

```
static Proc * searchmatch (const char * channel, Proc ** list) {
Proc * node = * list;
if (node == NULL) return NULL; /* the list is empty? * /
do {
if (strcmp (channel, node-> channel) == 0) {/ * found? * /
/ * remove a node from the list * /
if (* list == node) / * is this the first item in the list? * /
* list = (node-> next == node)? NULL: node-> next;
node-> previous-> next = node-> next;
node-> next-> previous = node-> previous;
return node;
}
node = node-> next;
} while (node! = * list);
return NULL; / * not found * /
}
```

The last helper function defined in listing ge 31.2, called when the process cannot find the friend it needs

gogo process. In this case, the process connects itself to the end of the
the corresponding list and waits until another process finds
and will not wake him up. (The loop around pthread_cond_wait protects against
random wakes that are possible on POSIX threads.) When
process wakes up another process, then it sets the channel field to
the awakened process to NULL . So if p-> channel is not equal
NULL , it means that no other process woke up this
process, so you have to wait further.

Listing 31.2. Function for adding a process to the waiting list

```
static void waitonlist (lua_State * L, const char * channel,
Proc ** list) {
Proc * p = getself (L);
/ * connect yourself to the end of the list * /
if (* list == NULL) {/ * is the list empty? * /
* list = p;
p-> previous = p-> next = p;
}
else {
p-> previous = (* list) -> previous;
p-> next = * list;
p-> previous-> next = p-> next-> previous = p;
}
p-> channel = channel;
do {/ * expects a conditional variable * /
pthread_cond_wait (& p-> cond, & kernel_access);
} while (p-> channel);
}
```

Now with these helper functions we can
write send and receieve (Listing 31.3). Send function starts
with checking the channel. Then it closes the mutex and looks for the matching
the recipient. If she finds him, then she carries her
values to this recipient, marks the recipient as ready to receive
fullness and wakes him up. Otherwise, she waits herself. Upon completion
Upon doing this, it opens a mutex and returns to Lua.
The receieve function is similar, but it must return everything received.
values.

Listing 31.3. Functions for sending and receiving messages

```
static int ll_send (lua_State * L) {
Proc * p;
const char * channel = luaL_checkstring (L, 1);
pthread_mutex_lock (& kernel_access);
p = searchmatch (channel, & waitreceive);
if (p) {/ * found a matching recipient? * /
movevalues (L, p-> L); / * transfer values to recipient * /
```

```
p-> channel = NULL; / * mark recipient * /
pthread_cond_signal (& p-> cond); / * wake him up * /
}
else
waitonlist (L, channel, & waitsend);
pthread_mutex_unlock (& kernel_access);
return 0;
}
static int ll_receive (lua_State * L) {
Proc * p;
const char * channel = luaL_checkstring (L, 1);
lua_settop (L, 1);
pthread_mutex_lock (& kernel_access);
p = searchmatch (channel, & waitsend);
if (p) {/ * found a matching recipient? * /
movevalues (p-> L, L); / * transfer values to recipient * /
p-> channel = NULL; / * mark recipient * /
pthread_cond_signal (& p-> cond); / * wake him up * /
}
else
waitonlist (L, channel, & waitreceive);
pthread_mutex_unlock (& kernel_access);
/ * return all values from the stack except the pipe * /
```

Lua states

```
return lua_gettop (L) - 1;
}
```

Now let's see how to create new processes. New
The process needs a new POSIX thread, and the new thread needs a body to execution. We will define this body later; here is the
totype:

```
static void * ll_thread (void * arg);
```

Listing 31.4. Function for creating a new process

```
static int ll_start (lua_State * L) {
pthread_t thread;
const char * chunk = luaL_checkstring (L, 1);
lua_State * L1 = luaL_newstate ();
if (L1 == NULL)
luaL_error (L, “unable to create new state”);
if (luaL_loadstring (L1, chunk)! = 0)
luaL_error (L, “error starting thread:% s”,
lua_tostring (L1, -1));
if (pthread_create (& thread, NULL, ll_thread, L1)! = 0)
luaL_error (L, “unable to create new thread”);
pthread_detach (thread);
return 0;
```

```
}
```

To create and launch a new process, the system needs to create a new lua state, start new thread, compile passed
block, call it and finally release its resources. Original thread performs the first three tasks and the new thread does the rest. (For to simplify error handling, the system starts a new thread after how she successfully compiled the given block.)
The ll_start function creates a new process (Listing 31.4). This the function creates a new Lua L1 state and compiles the given block in this new state. In case of an error, she informs initial states of L . Then she creates a new thread (with pthread_create) with the body ll_thread , passing the new state to L1 as a body argument. The pthread_detach call tells the system that we not expecting a definitive answer from this thread.

Listing 31.5. A body for new threads

```
int luaopen_lproc (lua_State * L);
static void * ll_thread (void * arg) {
lua_State * L = (lua_State *) arg;
luaL_openlibs (L); / * open standard libraries * /
luaL_requiref (L, “lproc”, luaopen_lproc, 1);
lua_pop (L, 1);
if (lua_pcall (L, 0, 0, 0)! = 0) / * call main chunk * /
fprintf (stderr, “thread error:% s”, lua_tostring (L, -1));
pthread_cond_destroy (& getself (L) -> cond);
lua_close (L);
return NULL;
}
```

The body of each new thread is the ll_thread function (the Thing 31.5). It gets its Lua state (created by ll_start) with an already compiled block on the stack. A new thread opens the stan- free Lua libraries, opens the lproc library and then calls has its own block. At the end, it releases its conditional variable (which Thoraya was created luaopen_lproc) and closes his fortune Lua.
Note the use of luaL_require for that- would open lproc 1 . This function is somewhat equivalent to require , but instead of searching for the bootloader, it uses the specified function (in our this is luaopen_lproc) to open the library. Pos-

before calling the opening function luaL_requiref registers a re-
the result is in the package.loaded table . If its last parameter is
true, then it also registers the library in the corresponding glo-
a point variable (in our case lproc).
The last function in our module, exit , is very simple:

```
static int ll_exit (lua_State * L) {
pthread_exit (NULL);
return 0;
}
```

Only the main process needs to call this function when it
will complete execution in order not to interrupt immediately
completing the entire program.
Our final step is to determine the opening
functions for the lproc module . This function is luaopen_lproc (fox-
Thing 31.6) must register module functions, but it also
should create and initialize the Proc structure of the current
process.
As I said earlier, this process definition in Lua is very simple.
There are an infinite number of improvements you can make
lick. Here I want to briefly discuss some of them.
The first obvious improvement would be to replace linear search
a process waiting on the specified channel. A beautiful alternative
would be using a hash table to find the channel and using
creation of independent waiting lists for each channel.
Another improvement relates to the efficiency of process creation.
sa. Creating a new state in Lua is a very fast operation.
However, opening all standard libraries is no longer so fast, and
most processes will likely not need all the standard
new libraries. We can avoid the price associated with opening
libraries, by pre-registering libraries, as we discussed
or in section 15.1. When using this approach, instead of calling
luaL_requiref for each standard library, we simply
We place the function that opens the library into the package.
preload . If the process calls require ”lib” , then and only then
yes require will call the appropriate function in order to
cover the library. Function registerlib (listing 31.7) performs
this registration.

Listing 31.6. Opening function for lproc module

```
static const struct luaL_reg ll_funcs [] = {
{“Start”, ll_start},
```

```
{“Send”, ll_send},
{“Receive”, ll_receive},
{“Exit”, ll_exit},
{NULL, NULL}
};
int luaopen_lproc (lua_State * L) {
/ * create your own control block * /
Proc * self = (Proc *) lua_newuserdata (L, sizeof (Proc));
lua_setfield (L, LUA_REGISTRYINDEX, “_SELF”);
self-> L = L;
self-> thread = pthread_self ();
self-> channel = NULL;
pthread_cond_init (& self-> cond, NULL);
luaL_register (L, “lproc”, ll_funcs); / * open library * /
return 1;
}
```

Listing 31.7. Registration of libraries on request

```
static void registerlib (lua_State * L, const char * name,
lua_CFunction f) {
lua_getglobal (L, “package”);
lua_getfield (L, -1, “preload”); / * get 'package.preload' * /
lua_pushcfunction (L, f);
lua_setfield (L, -2, name); / * package.preload [name] = f * /
lua_pop (L, 2); / * pop 'package' and 'preload' off the stack * /
static void openlibs (lua_State * L) {
luaL_requiref (L, “_G”, luaopen_base, 1);
luaL_requiref (L, “package”, luaopen_package, 1);
lua_pop (L, 2); / * remove the results of previous calls * /
registerlib (L, “io”, luaopen_io);
registerlib (L, “os”, luaopen_os);
registerlib (L, “table”, luaopen_table);
registerlib (L, “string”, luaopen_string);
registerlib (L, “math”, luaopen_math);
registerlib (L, “debug”, luaopen_debug);
}
```

It is always a good idea to open up the main library. You
you also need a package library, otherwise you won't
be able to use require to load other libraries. (You
don't even get the package.preload table .) All other libraries are
ki can be optional. So instead of calling luaL_open-
libs we will substitute our own openlibs function (also
shown in Listing 31.7) when creating new states. When
the process will need any of these libraries, it will explicitly require it,
and require will call the corresponding luaopen_ * function .
Other improvements include primitives for communication
tion. For example, it would be helpful to set limits on how long

lproc.send and lproc.receive may wait. In particular, the limit of expected a denier of zero will make these functions non-blocking. In threads POSIX we can implement this functionality with pthread_cond_timedwait .

Exercises

Exercise 31.1. As we have seen, if a function calls lua_yield (version without continuation function), control passed to the function that called it when the thread is again will continue its execution. What are the meanings of the caller will the function get how the results of this call?

Exercise 31.2. Modify the lproc library so that it could send other basic types such as booleans values and numbers. (*Hint* : you only need to change function movevalues .)

Exercise 31.3. In the lproc library, implement a non-blocking the send function .

Chapter 32
Memory management

Lua dynamically allocates all of its data structures. All these structures
tours grow dynamically as needed and over time decrease
change their size or disappear.
Lua is strict about its memory usage. When we for-
we hide the state of Lua, then Lua explicitly frees all its memory. Bo-
Moreover, all objects inside Lua are subject to garbage collection: not only
to tables and rows, but also functions, threads and modules (since
they are actually tables).
The way Lua manages memory is comfortable for most
applications. However, for some applications you may need -
adaptation, for example, to work in a limited space
memory space or to reduce garbage collector delays
to a minimum. Lua allows such adaptations to be done right away
on two levels. At the bottom level, we can define a function to use
used to allocate memory. At a higher level, we can
let's set some parameters to control the garbage collector
or we can even take direct control of the garbage collector.
In this chapter, we'll cover both of these options.

32.1. Function to highlight memory

The Lua core does not assume anything about memory allocation. To highlight it does not call malloc or realloc . Instead of this it performs all its allocation and deallocation of memory through single *out a function* (allocation function), which user must provide when creating Lua state.
The luaL_newstate function we used to create Lua states is a helper function that creates a Lua state with a default highlighting function. This function defaults to standard malloc-realloc-free from the C standard library, which should be enough for common applications. However, it is very easy to gain control. over memory allocation, creating your state using the function lua_newstate :

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

This function takes two arguments: the highlighting function and *the user data* (user data). The state created by this way, performs all allocation and deallocation of memory when using function calls f . (Even the structure lua_State vyde- is done with f .)
The type of the lua_Alloc allocating function is defined as follows. at once:

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

The first parameter is the user data that we will provide delivered lua_newstate ; the second parameter is the address of the block that we want to free or resize it; the third parameter is the original size of this block, and the fourth parameter is the request desired block size.
Lua guarantees that if ptr is not NULL then it was previously issued linen with osize size .

Lua uses NULL for zero-sized blocks. When nsize
is zero, then the function must free the block at the address ptr and ver-
a NULL string that corresponds to the requested block size.
When ptr is NULL , the function should allocate and return the block
a given size; if she cannot allocate a block of a given time
measure, then it must return NULL . If ptr is both NULL and nsize
is zero, then the function does nothing and returns NULL .
Finally, when ptr is both non- NULL and nsize is non-zero, the function
the tion should re-allocate this block (like realloc) and return a new one
address (which may be the same as the original address, or may differ
from him). Again, in case of an error, the function should return
NULL . Lua assumes that an allocating function is always successful.
works when the new size is less than or equal to the old size.
(Lua shrinks some structures during garbage collection and not in
able to handle errors correctly at this time.)
Function for allocating memory
The standard highlighting function used by luaL_newstate is
looks like this (taken from lauxlib.c file):

```
void * l_alloc (void * ud, void * ptr, size_t osize, size_t nsize) {
if (nsize == 0) {
free (ptr);
return NULL;
}
else
return realloc (ptr, nsize);
}
```

She believes that free (NULL) does nothing and that the call
realloc (NULL, size) is equivalent to malloc (size) . This guarantees-
xia ANSI C.
You can get an emitting function for a given state-
lua with lua_getallocf :

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

If ud is not NULL , then the function will set * ud to user values.
data used for this emitting function
tion. You can change the allocating function for Lua state
by calling lua_setallocf :

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

Keep in mind that the new highlighting function must be in
able to free blocks allocated by the old function. More often
the whole new highlighting function is just a wrapper over the old
function, for example, to track secretions or synchronize

heap access control.
Internally, Lua does not cache free blocks for reuse.
education. It assumes that the highlighting function does it,
many good memory allocation functions do this. Lua is not
tries to minimize memory fragmentation. Research on
show fragmentation is more the result of bad design
memory allocation than program behavior; good features for
memory allocations do not create strong fragmentation.
It is quite difficult to make a good highlighting function, but
sometimes you can try it. For example Lua gives you
the old size of any block when it is freed or changed
its size. Accordingly, a specialized emitting function
it is not necessary to store information about the block size somewhere, thus
reducing the amount of memory required for each block.
Another case where you can improve memory allocation is
case of multi-strand systems. Such systems usually require syn-
timing to allocate memory, since they use global
ny resource (memory). However, accessing Lua state must also
be synchronized - or better yet, limited to just one
thread, as in our lproc implementation in Chapter 31. So if each
before Lua state will allocate memory from its own memory pool-
ty, you can remove the explicit synchronization requirement.

32.2. Garbage collector

Prior to version 5.0, Lua used a simple *mark*- type garbage *collector*
and-sweep . This garbage collector is sometimes called *collector-stop-*
vi-world . This means that from time to time Lua will stop interpreting
run the main program to complete a complete build cycle
garbage. Each such cycle consists of three phases: *mark* ,
clean and *sweep* .
Lua starts the mark phase by marking it as live.
the root set that includes all objects to which Lua
has direct access: registry and main thread. Any object that
is stored in a living object, is reachable by the program and therefore also
tosses about as if alive. The tagging phase ends when all the
Objects are marked as live.
Before starting the sweeping phase, Lua performs a cleanup phase,

which is related to finalizers and weak tables. First of all,
it traverses all objects marked for finalization, looking for
unmarked objects. These objects are marked as alive (re-
bans) and placed on a separate list for use on
finalization stage. Second, Lua bypasses its weak tables and
removes from them all elements where either the key or the value itself is not marked.
The sweeping phase bypasses all Lua objects. (In order for this
it was possible that Lua kept all created objects with a coherent list
ke.) If the object is not marked as alive, then it is deleted. Otherwise
Lua unchecks it to prepare for the next loop.
During this phase, Lua also invokes object finalizers, which
some were collected during the cleaning phase.
Since version 5.1, Lua uses an incremental collector
garbage. This collector follows the same steps as the old one, but
for this he does not need to "stop the world." Instead, he worked

works with the interpreter. Every time the interpreter
allocates some memory, the garbage collector executes
small step. This means that while the garbage collector is running
works, the interpreter can change the visibility of the object. For
to ensure that the garbage collector works correctly, some
ry operations in the interpreter have special barriers, which
rye detect dangerous changes and correct the markings accordingly
corresponding objects.